# 4D Language Reference


- Introduction
- Language definition
- Debugging
- 4D Environment
- 4D Write Pro
- Arrays
- Backup
- BLOB
- Boolean
- Cache Management
- Communications
- Compiler
- Data Entry
- Database Methods
- Date and Time
- Design Object Access
- Drag and Drop
- Entry Control
- Form Events
- Forms
- Formulas
- Graphs
- Hierarchical Lists
- HTTP Client
- Import and Export
- Interruptions
- JSON
- Language
- LDAP
- List Box
- Math
- Menus
- Messages
- Named Selections
- Objects (Forms)
- Objects (Language)
- On a Series
- Operators
- Pasteboard
- PHP
- Pictures
- Printing
- Process (Communications)
- Process (User Interface)
- Processes
- Queries
- Quick Report
- Record Locking
- Records
- Relations
- Resources

- Secured Protocol
- Selection
- Sets
- Spell Checker
- SQL
- String
- Structure Access
- Styled Text
- Subrecords
- SVG
- System Documents
- System Environment
- Table
- Tools
- Transactions
- Triggers
- User Forms
- User Interface
- Users and Groups
- Variables
- Web Area
- Web Server
- Web Services (Client)
- Web Services (Server)
- Windows
- XML
- XML DOM
- XML SAX
- List of constant themes
- Error Codes
- Character Codes
- What's new
- Obsolete commands
- Alphabetical list of commands

# Introduction


- Copyrights and Legal notices
- Preface
- Introduction
- Building a 4D Application

## Copyrights and Legal notices

---

# Preface

4D has its own programming language. This built-in language, consisting of more than 1000 commands, makes 4D a powerful development tool for database applications on desktop computers. You can use the 4D language for many different tasks—from performing simple calculations to creating complex custom user interfaces. For example, you can:

- Programmatically access any of the record management editors (order by, query, and so on),
- Create and print complex reports and labels with the information from the database,
- Communicate with other devices,
- Manage documents,
- Import and export data between 4D databases and other applications,
- Incorporate procedures written in other languages into the 4D programming language.

The flexibility and power of the 4D programming language make it the ideal tool for all levels of users and developers to accomplish a complete range of information management tasks. Novice users can quickly perform calculations. Experienced users without programming experience can customize their databases. Experienced developers can use this powerful programming language to add sophisticated features and capabilities to their databases, including file transfer and communications. Developers with programming experience in other languages can add their own commands to the 4D language.

The 4D programming language is expanded when any of the 4D modules are added to the application. Each module includes language commands that are specific to the capabilities they provide.

## About the Manuals

The manuals described here provide a guide to the features of both 4D and 4D Server. The only exception is the 4D Server Reference, which describes features exclusive to 4D Server.

- The Language Reference is a guide to using the 4D language. Use this manual to learn how to customize your database by incorporating 4D commands and functions.
- The Design Reference provides detailed descriptions of the editors and tools available in this environment.
- The Self-training manual leads you through example lessons in which you create and use a 4D database. These examples provide hands-on experience and help you become familiar with the concepts and features of 4D and 4D Server.
- The 4D Server Reference, which is included only in the 4D Server package, is a guide to managing multi-user databases with 4D Server.

## About this Manual

This manual describes the 4D language. It assumes that you are familiar with terms such as table, field, and form. Before you read this manual, you should:

- Use the Self-training manual to work through the database example.
- Begin creating your own databases, referring to the Design Reference manual when necessary.
- Increase your knowledge by studying that numerous demo and example databases that are available on the 4D Web site (*http://www.4d.com*).

## Writing conventions

In this manual, several writing conventions are used:

- Following the example of the 4D Method editor, commands are written in all caps using special characters, e.g.: **CLOSE DOCUMENT**. Functions (commands that return a value) start with a capital letter and continue in lower case, e.g.: **Change string**.
- In the command syntax, the { } characters (braces) indicate optional parameters. For example, **SET DEFAULT CENTURY (century{; pivotYear})** means that the *pivotYear* parameter may be omitted when calling the command.

- In the command syntax, the | character indicates an alternative. For example, **Table (tableNum | aPtr)** indicates that the function accepts either a table number or a pointer as parameter.
- In certain examples in this documentation, a line of code may be continued onto the following line(s) due to lack of space. However, you should type these examples as a single line of code without using carriage returns.

## Where to go from here?

If you are reading this manual for the first time, read the **Introduction** section.

# Introduction

This topic introduces you to the 4D programming language. The following topics are discussed:

- What the language is and what it can do for you,
- How you will use methods,
- How to develop an application with 4D.

These topics are covered here in general terms; they are covered in greater detail in other sections.

## What is a Language?

The 4D language is not very different from the spoken language we use every day. It is a form of communication used to express ideas, inform, and instruct. Like a spoken language, 4D has its own vocabulary, grammar, and syntax; you use it to tell 4D how to manage your database and data.

You do not need to know everything in the language in order to work effectively with 4D. In order to speak, you do not need to know the entire English language; in fact, you can have a small vocabulary and still be quite eloquent. The 4D language is much the same—you only need to know a small part of the language to become productive, and you can learn the rest as the need arises.

## Why Use a Language?

At first it may seem that there is little need for a programming language in 4D. In the Design environment, 4D provides flexible tools that require no programming to perform a wide variety of data management tasks. Fundamental tasks, such as data entry, queries, sorting, and reporting are handled with ease. In fact, many extra capabilities are available, such as data validation, data entry aids, graphing, and label generation.

Then why do we need a 4D language? Here are some of its uses:

- Automate repetitive tasks: These tasks include data modification, generation of complex reports, and unattended completion of long series of operations.
- Control the user interface: You can manage windows and menus, and control forms and interface objects.
- Perform sophisticated data management: These tasks include transaction processing, complex data validation, multi-user management, sets, and named selection operations.
- Control the computer: You can control serial port communications, document management, and error management.
- Create applications: You can create easy-to-use, customized databases that run in the Application environment.
- Add functionality to the built-in 4D Web Services: Create dynamic HTML pages in addition to those automatically translated from forms by 4D.

The language lets you take complete control over the design and operation of your database. 4D provides powerful “generic” editors, but the language lets you customize your database to whatever degree you require.

## Taking Control of Your Data

The 4D language lets you take complete control of your data in a powerful and elegant manner. The language is easy enough for a beginner, and sophisticated enough for an experienced application developer. It provides smooth transitions from built-in database functions to a completely customized database.

The commands in the 4D language provide access to the standard record management editors. For example, when you use the **QUERY** command, you are presented with the Query Editor (which can be accessed in the Design mode using the **Query** command in the **Records** menu. You can tell the **QUERY** command to search for explicitly described data. For example, **QUERY** *([People];[People]Last Name="Smith")* will find all the people named Smith in your database.

The 4D language is very powerful—one command often replaces hundreds or even thousands of lines of code written in traditional computer languages. Surprisingly enough, with this power comes simplicity—commands have plain English names. For example, to perform a query, you use the **QUERY** command; to add a new record, you use the **ADD RECORD** command.

The language is designed for you to easily accomplish almost any task. Adding a record, sorting records, searching for data, and similar operations are specified with simple and direct commands. But the language can also control the serial ports, read disk documents, control sophisticated transaction processing, and much more.

The 4D language accomplishes even the most sophisticated tasks with relative simplicity. Performing these tasks without using the language would be unimaginable for many.
Even with the language's powerful commands, some tasks can be complex and difficult. A tool by itself does not make a task possible; the task itself may be challenging and the tool can only ease the process. For example, a word processor makes writing a book faster and easier, but it will not write the book for you. Using the 4D language will make the process of managing your data easier and will allow you to approach complicated tasks with confidence.

## Is it a "Traditional" Computer Language?

If you are familiar with traditional computer languages, this section may be of interest. If not, you may want to skip it.

The 4D language is not a traditional computer language. It is one of the most innovative and flexible languages available on a computer today. It is designed to work the way you do, and not the other way around.

To use traditional languages, you must do extensive planning. In fact, planning is one of the major steps in development. 4D allows you to start using the language at any time and in any part of your database. You may start by adding a method to a form, then later add a few more methods. As your database becomes more sophisticated, you might add a project method controlled by a menu. You can use as little or as much of the language as you want. It is not "all or nothing," as is the case with many other databases.

Traditional languages force you to define and pre-declare objects in formal syntactic terms. In 4D, you simply create an object, such as a button, and use it. 4D automatically manages the object for you. For example, to use a button, you draw it on a form and name it. When the user clicks the button, the language automatically notifies your methods.

Traditional languages are often rigid and inflexible, requiring commands to be entered in a very formal and restrictive style. The 4D language breaks with tradition, and the benefits are yours.

## Methods are the Gateway to the Language

A method is a series of instructions that causes 4D to perform a task. Each line of instruction in a method is called a statement. Each statement is composed of parts of the language.

Because you have already worked through the Quickstart tutorials (you did go through Quickstart, didn't you?), you have already written and used methods.

You can create five types of methods with 4D:

- **Object Methods**: Usually short methods used to control form objects.
- **Form Methods**: Manage the display or printing of a form.
- **Table Methods/Triggers**: Used to enforce the rules of your database.
- **Project methods**: Methods that are available for use throughout your database. For example, methods that can be attached to menus.
- **Database methods**: Execute initializations or special actions when a database is opened or closed, or when a Web browser connects to your database published as a Web Server on Internet an Intranet.

The following sections introduce each of these method types and give you a feel for how you can use them to automate your database.

### Getting started with object methods

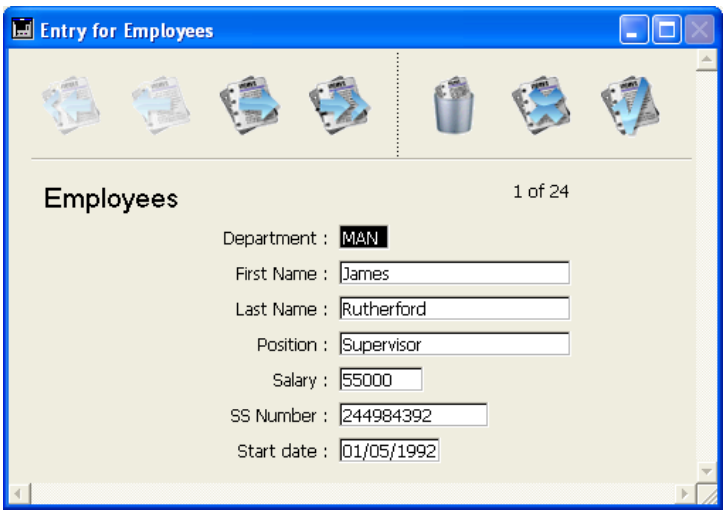
Any form object that can perform an action (that is, any active object) can have a method associated with it. An object method monitors and manages the active object during data entry and printing. A object method is bound to its active object even when the object is copied and pasted. This allows you to create reusable libraries of scripted objects. The object method takes control exactly when needed.

Object methods are the primary tools for managing the user interface, which is the doorway to your database. The user interface consists of the procedures and conventions by which a computer communicates with the user. The goal is to make the user interface of your database as simple and easy to use as possible. The user interface should make interaction with the computer a pleasant process, one that the user enjoys or does not even notice.

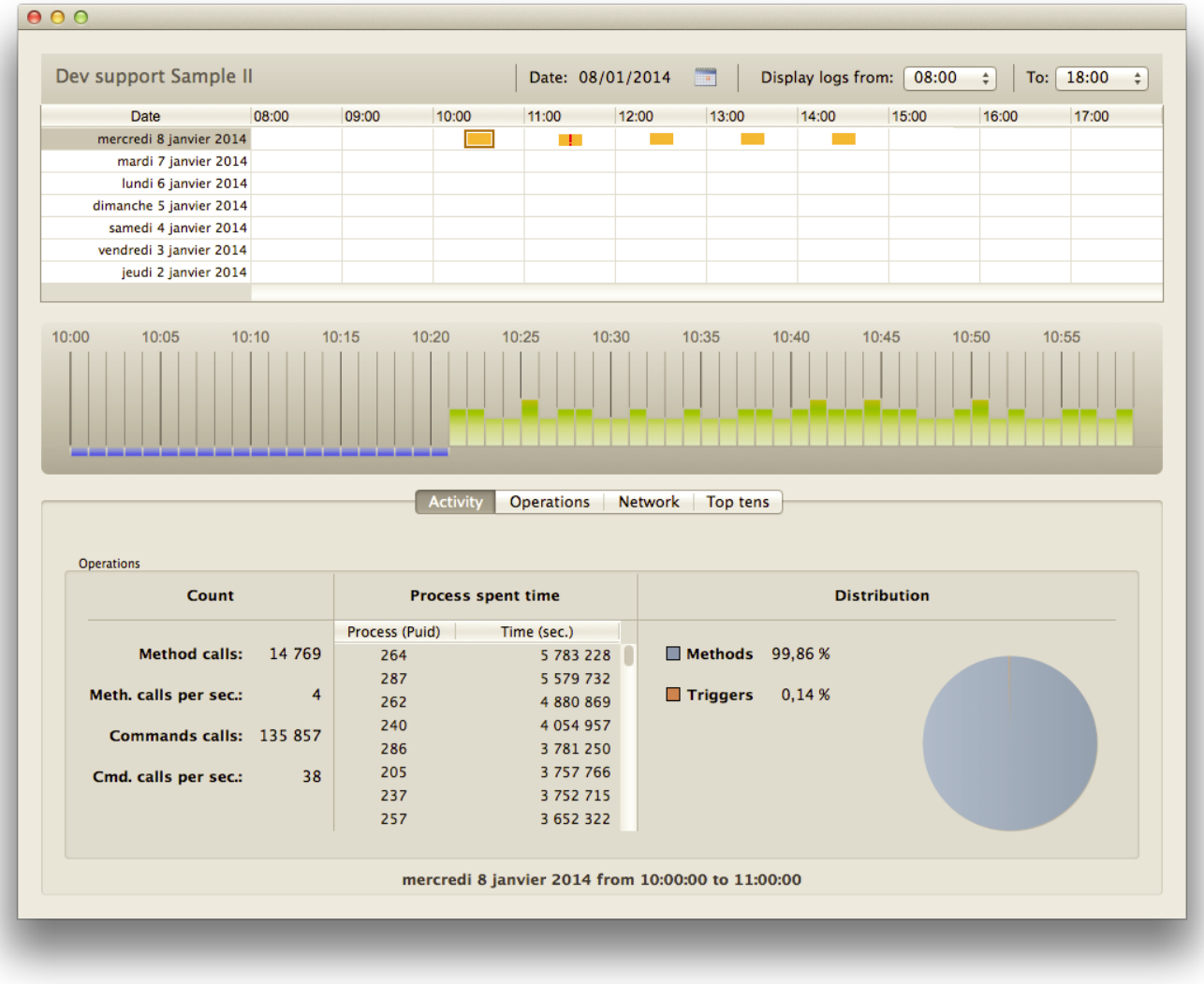
There are two basic types of active objects in a form:

- Those for entering, displaying, and storing data; such as fields and subfields
- Those for control; such as enterable areas, buttons, scrollable areas, hierarchical lists, and meters

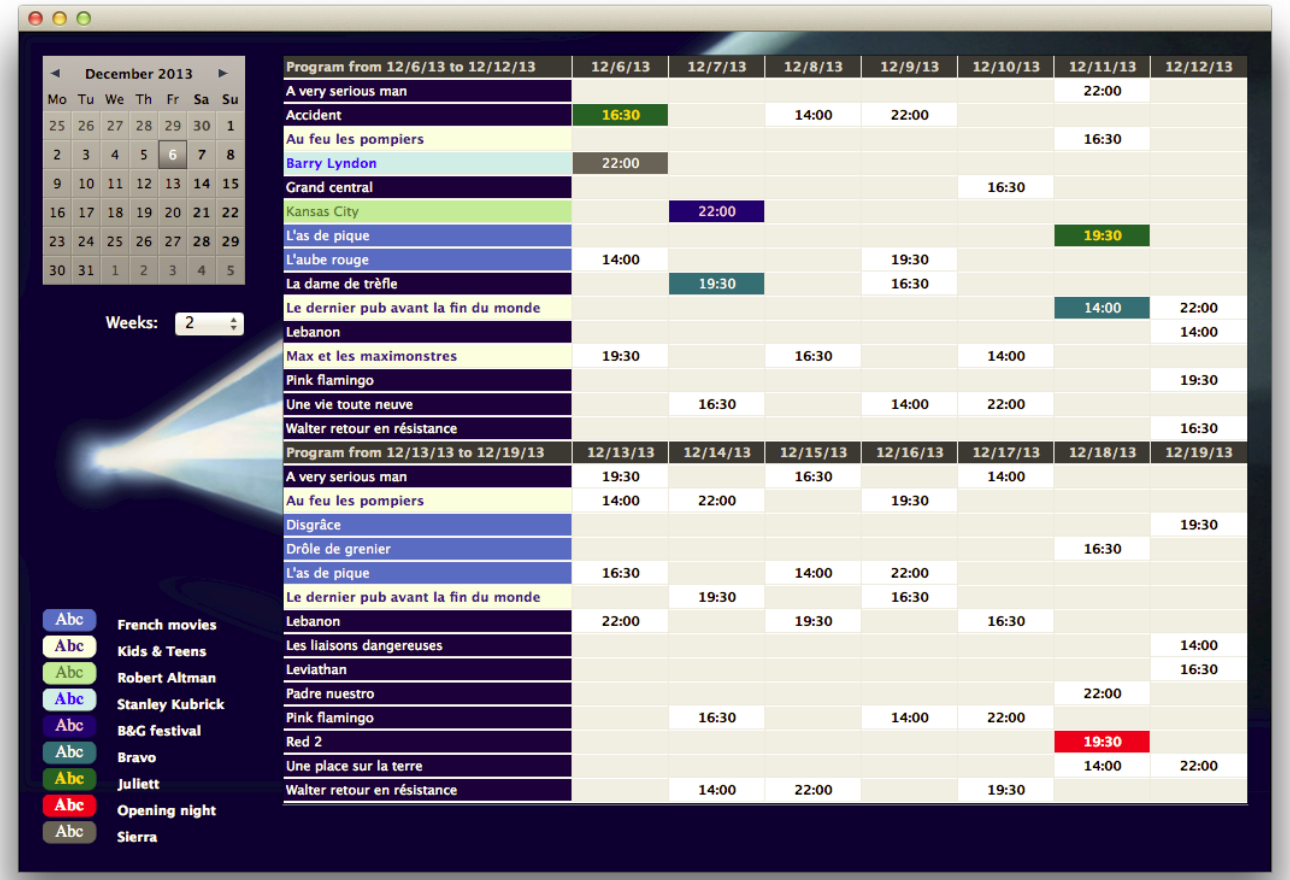
4D enables you to build classic forms, such as the one shown here:




You can also build forms with multiple graphic controls, such as this one:




You can even build forms that incorporate a graphical flair limited only by your imagination:

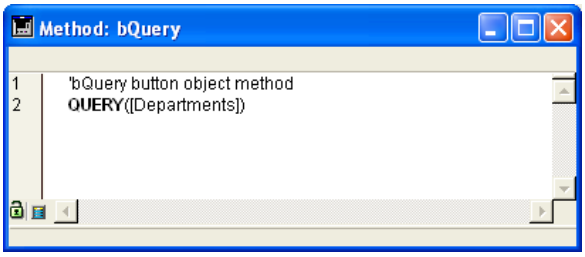



| Program from 12/6/13 to 12/12/13 | 12/6/13 | 12/7/13 | 12/8/13 | 12/9/13 | 12/10/13 | 12/11/13 | 12/12/13 |
|---|---|---|---|---|---|---|---|
| A very serious man | | | | | | 22:00 | |
| Accident | 16:30 | | 14:00 | 22:00 | | | |
| Au feu les pompiers | | | | | | 16:30 | |
| Barry Lyndon | 22:00 | | | | | | |
| Grand central | | | | | 16:30 | | |
| Kansas City | | 22:00 | | | | | |
| L'as de pique | | | | | | 19:30 | |
| L'aube rouge | 14:00 | | | 19:30 | | | |
| La dame de trèfle | | 19:30 | | 16:30 | | | |
| Le dernier pub avant la fin du monde | | | | | | 14:00 | 22:00 |
| Lebanon | | | | | | | 14:00 |
| Max et les maximonstres | 19:30 | | 16:30 | | 14:00 | | |
| Pink flamingo | | | | | | | 19:30 |
| Une vie toute neuve | | 16:30 | | 14:00 | 22:00 | | |
| Walter retour en résistance | | | | | | | 16:30 |
| **Program from 12/13/13 to 12/19/13** | **12/13/13** | **12/14/13** | **12/15/13** | **12/16/13** | **12/17/13** | **12/18/13** | **12/19/13** |
| A very serious man | 19:30 | | 16:30 | | 14:00 | | |
| Au feu les pompiers | 14:00 | 22:00 | | 19:30 | | | |
| Disgrâce | | | | | | | 19:30 |
| Drôle de grenier | | | | | | 16:30 | |
| L'as de pique | 16:30 | | 14:00 | 22:00 | | | |
| Le dernier pub avant la fin du monde | | 19:30 | | 16:30 | | | |
| Lebanon | 22:00 | | 19:30 | | 16:30 | | |
| Les liaisons dangereuses | | | | | | | 14:00 |
| Leviathan | | | | | | | 16:30 |
| Padre nuestro | | | | | | 22:00 | |
| Pink flamingo | | 16:30 | | 14:00 | 22:00 | | |
| Red 2 | | | | | | 19:30 | |
| Une place sur la terre | | | | | | 14:00 | 22:00 |
| Walter retour en résistance | | 14:00 | 22:00 | | 19:30 | | |

Whatever your style in building forms, all active objects have built-in aids, like range checking and entry filters for data entry areas, and automatic actions for controls, menus, and buttons. Always use these aids before adding object methods. The built-in aids are similar to methods in that they remain associated with the active object and are active only when the active object is being used. You will typically use a combination of built-in aids and object methods to control the user interface.

An object method associated with an active object used for data entry typically performs a data-management task specific to the field or variable. The method can perform data validation, data formatting, or calculations. It may even get related information from other files. Some of these tasks can, of course, also be performed with the built-in data entry aids for objects. Use object methods when the task is too complex for the built-in data entry aids to manage. For more information about the built-in data entry aids, refer to the 4D Design Reference manual.

Object methods are also associated with active objects used for control, such as buttons. Active objects used for control are essential to navigating within your database. Buttons allow you to move from record to record, move to different forms, and add and delete data. These active objects simplify the use of a database and reduce the time required to learn it. Buttons also have built-in aids and, as with data entry, you should use these built-in aids before adding methods. Object methods enable you to add actions that are not built-in, to your controls. For example, the following window is the object method for a button that, when clicked, displays the Query editor.




As you become more proficient with scripts, you will find that you can create libraries of objects with associated methods. You can copy and paste these objects and their methods between forms, tables, and databases.

### Controlling forms with form methods

In the same way that object methods are associated with the active objects in a form, a form method is associated with a form. Each form can have one form method. A form is the means through which you can enter, view, and print your data. Forms allow you to present the data to the user in different ways. Through the use of forms, you can create attractive and

easy-to-use data entry screens and printed reports. A form method monitors and manages the use of an individual form both for data entry and for printing.

Form methods manage forms at a higher level than do object methods. Object methods are activated only when the object is used, whereas a form method is activated when anything in the form is used. Form methods are typically used to control the interaction between the different objects and the form as a whole.

As forms are used in so many different ways, it is informative to monitor what is happening while your form is in use. You use the various **form events** for this purpose. They tell you what is currently happening with the form. Each type of event (i.e., clicks, double-clicks, keystrokes...) enables or disables the execution of the form method as well as the object method of each object of the form.

For more information about form, objects, events and methods, refer to the description of the **Form event** command.

### Enforcing the rules of your database using the table methods/triggers

A Trigger is attached to a table; for this reason, it is also called a Table Method. Triggers are automatically invoked by the 4D database engine each you manipulate the records of a table (Add, Delete, Modify and Load). Triggers are methods that can prevent “illegal” operations with the records of your database. For example, in an invoicing system, you can prevent anyone from adding an invoice without specifying the customer to whom the invoice is billed. Triggers are a very powerful tool to restrict operations on a table as well as to prevent accidental data loss or tampering. You can write very simple triggers, then make them more and more sophisticated.

For detailed information, see the **Triggers** section.

### Using project method throughout the database

Unlike object methods, form methods, and triggers, which are all associated with a particular object, form, or table, project methods are available for use throughout your database. Project methods are reusable, and available for use by any other method. If you need to repeat a task, you do not have to write identical methods for each case. You can call project methods wherever you need them—from other project methods or from object or form methods. When you call a project method, it acts as if you had written the method at the location where you called it. Project methods called from other method are often referred to as “subroutines.”

There is one other way to use project methods—associating them with menu commands. When you associate a project method with a menu command, the method is executed when the menu is chosen. You can think of the menu command as calling the project method.

### Handling working sessions with database methods

In the same way object and form methods are invoked when events occur in a form, there are methods associated with the database which are invoked when a working session event occurs. These are the **database methods**. For example, each time you open a database, you may want to initialize some variables that will be used during the whole working session. To do so, you use the **On Startup database method**, automatically executed by 4D when you open the database.

For more information about Database Methods, see the **Database Methods** section.

## Developing Your Database

Development is the process of customizing a database using the language and other built-in tools.

By simply creating a database, you have already taken the first steps to using the language. All the parts of your database—the tables and fields, the forms and their objects, and the menus—are tied to the language. The 4D language “knows” about all of these parts of your database.

Perhaps your first use of the language is to add a method to a form object in order to control data entry. Later, you might add a form method to control the display of your form. As the database becomes more complex, you can add a menu bar with project methods to completely customize your database.

As with other aspects of 4D, development is a very flexible process. There is no formal path to take during development—you can develop in a manner with which you are comfortable. There are, of course, some general patterns in the process.

- Implementation: Implement your design in the Design environment.
- Testing: You try out the design and test each customized element using the Test Application command to launch the Application environment.
- Usage: When your database is fully customized, you launch it directly in the Application environment.
- Corrections: If you find errors, you return to the Design environment to fix them.

Special development support tools, hidden until needed, are built into 4D. As you use the language more frequently, you will find that these tools facilitate the development process. For example, the Method Editor catches typing errors and formats

your work; the Interpreter (the engine that runs the language) catches errors in syntax and shows you where and what they are; and the Debugger lets you monitor the execution of your methods to catch errors in design.

## Building Applications

By now you are familiar with the general uses of a database—data entry, searching, sorting, and reporting. You have performed these actions in the Design environment, using the standard menus and editors.

As you use a database, you perform some sequences of tasks repeatedly. For example, in a database of personal contacts, you might search for your business associates, order them by last name, and print a specific report each time information about them is changed. These tasks may not seem difficult, but they can certainly be time-consuming after you have done them 20 times. In addition, if you don't use the database for a couple of weeks, you may return to find that the steps used to generate the report are not so fresh in your mind. The steps in methods are chained together, so a single command automatically performs all the tasks linked to it. Consequently, you do not have to worry about the specific steps.

Applications have custom menus and perform tasks that are specific to the needs of the person using your database. An application is composed of all the pieces of your database: the structure, the forms, the object, form and project methods, the menus, and the passwords.

You can compile your databases and create stand-alone Windows and Macintosh applications. Compiling databases increases the execution speed of the language, protects your databases, and allows you to create applications that are completely independent. The integrated compiler also checks the syntax and the types of variables in methods for consistency.

An application can be as simple as a single menu that lets you enter people's names and print a report, or as complex as an invoicing, inventory, and control system. There are no limits to the uses of database applications. Typically, an application grows from a database used in the Design environment to a database controlled completely by custom menus and forms.

## Where to go from here?

- Developing applications can be as simple or complex as you like. For a quick overview about building a simple 4D application, see the **Building a 4D Application** section.
- If you are new to 4D, refer to the **Language definition** sections to learn about the basics of the 4D language: start with **Introduction to the 4D Language**.

# Building a 4D Application

An application is a database designed to fill a specific need. It has a user interface designed specifically to facilitate its use. The tasks that an application performs are limited to those appropriate for its purpose. Creating applications with 4D is smoother and easier than with traditional programming. 4D can be used to create a variety of applications, including:

- An invoice system
- An inventory control system
- An accounting system
- A payroll system
- A personnel system
- A customer tracking system
- A database shared over the Internet or an Intranet

It is possible that a single application could even contain all of these systems. Applications like these are typical uses of databases. In addition, the tools in 4D allow you to create innovative applications, such as:

- A document tracking system
- A graphic image management system
- A catalog publishing application
- A serial device control and monitoring system
- An electronic mail system (E-mail)
- A multi-user scheduling system
- A list such as a menu list, video collection, or music collection

An application typically can start as a database used in the Design environment. The database “evolves” into an application as it is customized. What differentiates an application is that the systems required to manage the database are hidden from the user. Database management is automated, and users use menus to perform specific tasks.

When you use a 4D database in the Design environment, you must know the steps to take to achieve a result. In an application, you use the Application environment, in which you need to manage all the aspects that are automatic in the Design Environment. These include:

- Table Navigation: The **List of Tables** window, the Last used tables command or the navigation buttons are not available to the user. You can use menu commands and methods to control navigation between tables.
- Menus: In the Application environment, you only have the default File menu with the Quit menu command, the Edit menu, the Mode and the Help menu (as well as the application menu under Mac OS). If the application requires more menus, you have to create and manage them using 4D methods or standard actions.
- Editors: The editors, such as the Query and Order By editors, are no longer automatically available in the Application environment. If you want to use them, you have to call them using 4D methods.

The following sections include examples showing how the language can automate the use of a database.

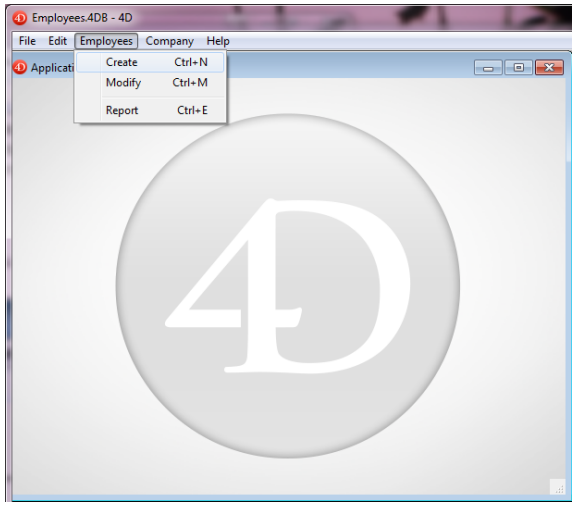
## Application Environment: an Example

Custom menus are the primary interface in an application. They make it easier for users to learn and use a database. Creating custom menus is very simple—you associate methods or automatic actions with each menu command (also called menu items) in the Menu editor.

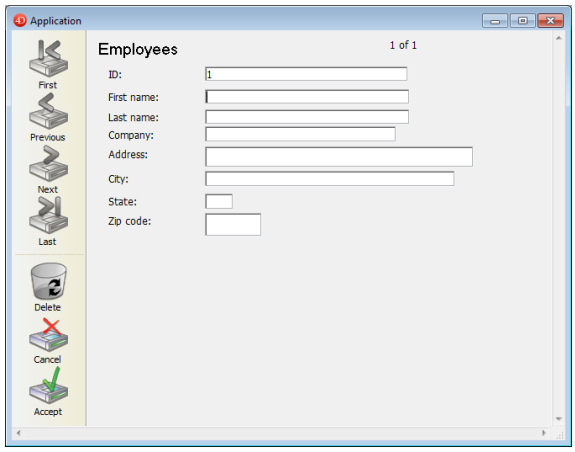
“The User's Perspective” section describes what happens when the user chooses a menu command. Next, “Behind the Scenes” describes the design work that made it happen. Although the example is simple, it should be apparent how custom menus make the database easier to use and learn. Rather than the “generic” tools and menu commands in the Design environment, the user sees only things that are appropriate to his or her needs.

### The User’s Perspective

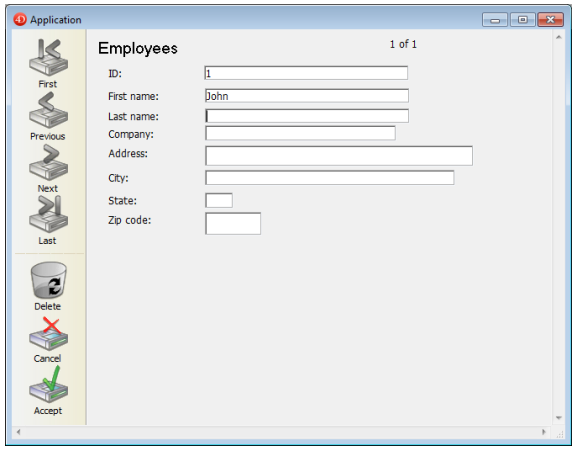
The user chooses a menu item called **Create** from the **Employees** menu to add a new person to the database.



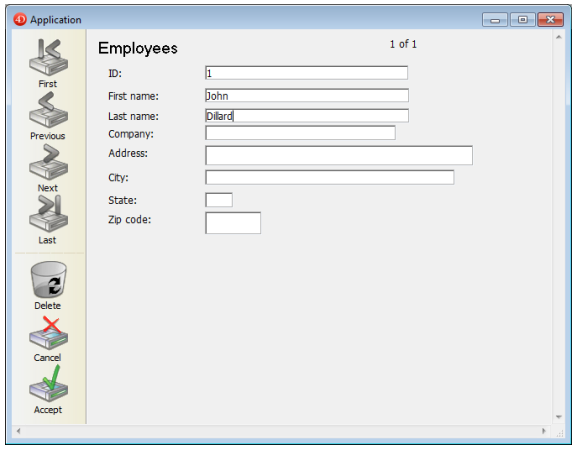

The Input form for the Employees table is displayed.




The user enters the person's first name and then tabs to the next field.

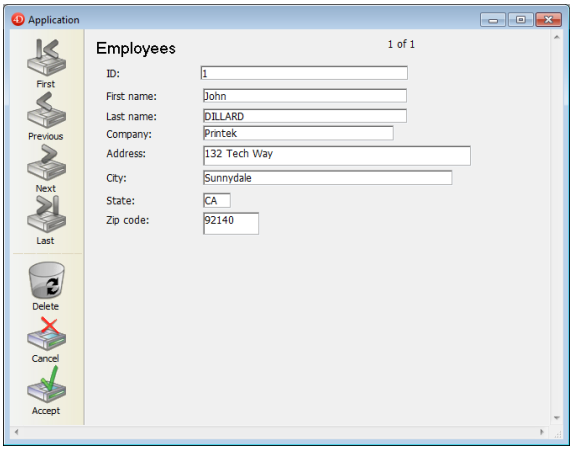



The user enters the person's last name.




The user tabs to the next field: the last name is converted to uppercase.

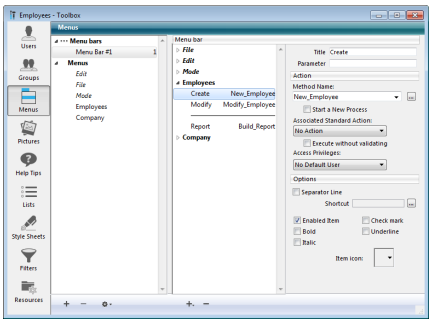
The user finishes entering the record and clicks the validation button (generally the last button in the button bar).



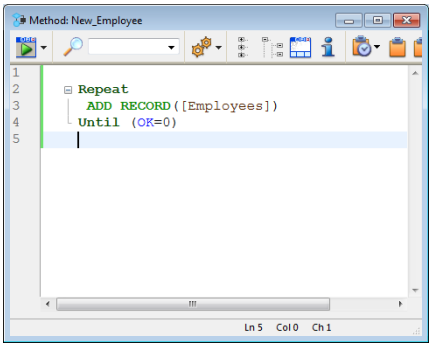

Another blank record appears, and the user clicks the Cancel button (the one with the "X") to terminate the "data entry loop." The user is returned to the menu bar.

### Behind the Scenes

The menu bar was created in the Design environment, using the Menu Bar Editor.



The menu item **New** has a project method named **New Employee** associated with it. This method was created in the Design environment, using the Method editor.




When the user chooses this menu item, the **New Employee** method executes:

```
Repeat
   ADD RECORD([Employees])
Until(OK=0)
```

The **Repeat...Until** loop with an **ADD RECORD** command within the loop acts just like the **New Record** menu item in the Design environment. It displays the input form to the user, so that he or she can add a new record. When the user saves the record, another new blank record appears. This **ADD RECORD** loop continues to execute until the user clicks the **Cancel** button.

When a record is entered, the following occurs:

- There is no method for the **First Name** field, so nothing executes.
- There is a method for the **Last Name** field. This Object Method was created in the Design environment, using the Form and Method editors. The method executes:

```
[Employees]Last Name:=Uppercase([Employees]Last Name)
```

This line converts the **Last Name** field to uppercase characters.

After a record has been entered, when the user clicks the Cancel button for the next one, the OK variable is set to zero, thus ending the execution of the **ADD RECORD** loop.

As there are no more statements to execute, the **New Employee** method stops executing and control returns to the menu bar.

## Comparing an Automated Task with the Actions to be performed in the Design environment

Let's compare the way a task is performed in the Design environment and the way the same task is performed using the language. The task is a common one:

- Find a group of records
- Sort them
- Print a report

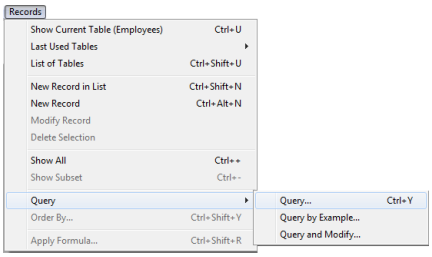
The next section, "Using a Database in the Design Environment," displays the tasks performed in the Design environment.

The following section, "Using the Built-in Editors within the Application environment," displays the same tasks performed in an application.

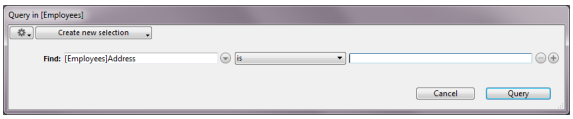
Note that although both methods perform the same task, the steps in the second section are automated using the language.

### Using a database in the Design environment

The user chooses **Query**>**Query...** in the **Records** menu.



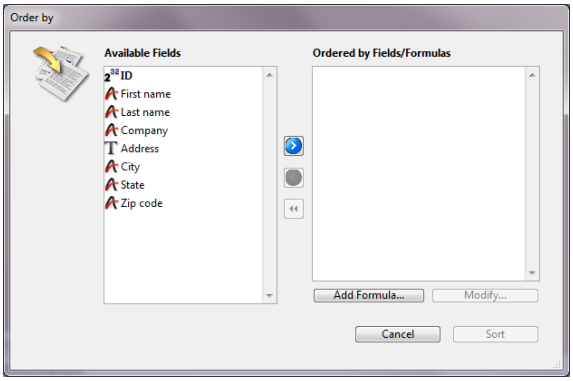

The **Query** editor is displayed.




The user enters the criteria and clicks the **Query** button. The search is performed.

The user chooses **Order by** from the **Records** menu.


The **Order By** editor is displayed.




The user enters the criteria and clicks the **Sort** button. The sort is performed.

Then, to print the records, these additional steps are required:

- The user chooses **Print** from the **File** menu.
- The **Choose Print Form** dialog box is displayed, because users need to know which form to print.

- The Printing dialog boxes are displayed. The user chooses the settings, and the report is printed.

### Using the built-in editors within the Application environment

Let's examine how this can be performed in the Application environment.

The User chooses **Report** from the Employees menu.

Even at this point, using an application is easier for the users—they did not need to know that querying is the first step!

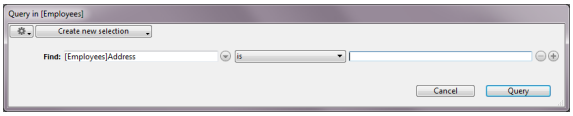
A method called **Build Report** is attached to the menu command; it looks like this:

```
QUERY([Employees])
ORDER BY([Employees])
FORM SET OUTPUT([Employees];"Report")
PRINT SELECTION([Employees])
```

The first line is executed:

```
QUERY([Employees])
```

The **Query** editor is displayed.



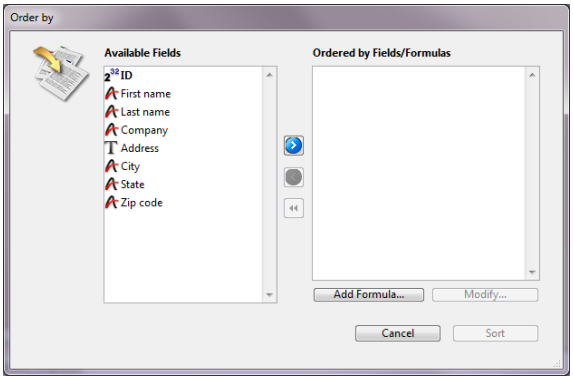

The user enters the criteria and clicks the **Query** button. The query is performed.

The second line of the **Build Report** method is executed:

```
ORDER BY([Employees])
```

Note that the user did not need to know that ordering the records was the next step.

The **Order By** Editor is displayed.




The user enters the criteria and clicks the **Sort** button. The sort is performed.

The third line of the **Build Report** method is executed:

```
FORM SET OUTPUT([Employees];"Report")
```

Once again, the user did not need to know what to do next; the method takes care of that.

The final line of the **Build Report** method is executed:

```
PRINT SELECTION([Employees])
```

The **Printing** dialog boxes are displayed. The User chooses the settings, and the report is printed.

## Automating the Application Further

The same commands used in the previous example can be used to further automate the database.

Let's take a look at the new version of the **Build Report** method.

The user chooses **Report** from the **Employees** menu. A method called **Build Report2** is attached to the menu command. It looks like this:

```
QUERY([Employees];[Employees]Company="Acme")
ORDER BY([Employees];[Employees]Last Name;>;[Employees]First Name;>)
FORM SET OUTPUT[Employees];"Report")
PRINT SELECTION([Employees];*)
```

The first line is executed:

```
QUERY([Employees];[Employees]Company="Acme")
```

The **Query** editor is not displayed. Instead, the query is specified and performed by the **QUERY** command. The user does not need to do anything.

The second line of the **Build Report2** method is executed:

```
ORDER BY([Employees];[Employees]Last Name;>;[Employees]First Name;>)
```

The **Order By** editor is not displayed, and the sort is immediately performed. Once again, no user actions are required.

The final lines of the **Build Report2** method are executed:

```
FORM SET OUTPUT[Employees];"Report")
PRINT SELECTION([Employees];*)
```

The **Printing** dialog boxes are not displayed. The **PRINT SELECTION** command accepts an optional asterisk (*) parameter that instructs the command to use the print settings that were in effect when the report form was created. The report is printed.

This additional automation saved the user from having to enter options in three dialog boxes. Here are the benefits :

- The query is automatically performed: users may select wrong criteria when making a query.
- The sort is automatically performed: users may select wrong criteria when defining a sort.
- The printing is automatically performed: users may select the wrong form to print.

## Help for Developing 4D Applications

As you develop a 4D application, you will discover many capabilities that you did not notice when you started. You can even augment the standard version of 4D by adding other tools and plug-ins to your 4D development environment.

### Plug-ins 4D

4D provides several plug-ins that can be used for increasing the capabilities of your 4D applications, including:

- **4D Write**: Word-processor
- **4D View**: Spreadsheet and list editor
- **4D Internet Commands** (built-in): Communication utilities via Internet.

- **4D ODBC Pro**: Connectivity via ODBC
- **4D for OCI**: Connectivity with ORACLE Call Interface

For more information, contact 4D or its Partners, or visit our Web site:

http://www.4d.com

### The 4D community and third party tools

There is a very active worldwide 4D community, composed of User Groups, Electronic Forums, and 4D Partners. 4D Partners produce **Third Party Tools**. You can suscribe to the user forum of 4D at the following address:

http://forums.4D.fr

The 4D community offers access to tips and tricks, solutions, information, and additional tools that will save you time and energy, and increase your productivity.

# Language definition


- Introduction to the 4D Language
- Constants
- Variables
- System Variables
- Pointers
- Identifiers
- Control Flow
- If...Else...End if
- Case of...Else...End case
- While...End while
- Repeat...Until
- For...End for
- Methods
- Project Methods

# Introduction to the 4D Language

The 4D language is made up of various components that help you perform tasks and manage your data.

- **Data types**: Classifications of data in a database. See discussion in this section as well as the detailed discussion in the section **Data Types**.
- **Variables**: Temporary storage places for data in memory. See detailed discussion in the section **Variables**.
- **Operators**: Symbols that perform a calculation between two values. See discussion in this section as well as the detailed discussion in the section **Operators** and its subsections.
- **Expressions**: Combinations of other components that result in a value. See discussion in this section.
- **Commands**: Built-in instructions to perform an action. All 4D commands, such as **ADD RECORD**, are described in this manual, grouped by theme; when necessary, the theme is preceded by an introductory section. You can use 4D Plug-ins to add new commands to your 4D development environment. For example, once you have added the 4D Write Plug-in to your 4D system, the 4D Write commands become available for creating and manipulating word-processing documents.
- **Predefined constants**: Constant values accessible by name. For example, XML DATA is a constant (value 6). Predefined constants allow writing more readable code. Constants are described with the commands that use them, and are fully listed in the **List of constant themes** section.
- **Methods**: Instructions that you write using all parts of the language listed here. See discussion in the section **Methods** and its subsections.

This section introduces **Data Types**, **Operators**, and **Expressions**. For the other components, refer to the sections cited above.

In addition:

- Language components, such as variables, have names called Identifiers. For a detailed discussion about identifiers and the rules for naming objects, refer to the section **Identifiers**.
- To learn more about array variables, refer to the section **Arrays**.
- To learn more about BLOB variables, refer to the section **BLOB Commands**.
- If you plan to compile your database, refer to the section **Compiler Commands** as well as the Design Reference manual of 4D.

## Language for commands and constants

Starting with 4D v15, 4D's Method editor uses the international "English-US" language by default, regardless of the 4D version or local system settings. This feature neutralizes any regional variations that might disrupt code interpretation between 4D applications (date formats for instance); and in French versions of 4D, commands and constants are now written in "English-US" as is already the case in other languages.

This default setting provides 4D developers with several advantages:

- It facilitates code sharing between developers, regardless of their country, regional settings, or the 4D version used. A 4D method can now be exchanged by simple copy/paste, or saved in a text file, with no compatibility issues.
- It also makes it possible to include 4D methods in source control tools, which often require exports to be independent from regional settings and languages.

This setting can be disabled using the "Use regional system settings" option in the 4D Preferences dialog box (see the **Methods Page**).

### Input principles in English-US

The English-US settings may have several effects on the way you write methods. This concerns code written in development mode as well as formulas in deployed applications. In this mode, code must comply with following rules:

- Decimal separators for real numbers must now be periods (".") in all versions (and not commas (",") as is the custom in French, for example).
- Date constants must now use the ISO format (!YYYY-MM-DD!) in all versions.

- Command and constant names must be in English (this change only concerns French versions of 4D, since this was already the case with other languages).

**Note:** The Method editor includes specific mechanisms that automatically fix incorrect entries if necessary.

The following table illustrates differences between code in 4D v15 (or higher) and in previous versions:

| | **Code sample in methods/formulas** |
|---|---|
| **4D v15 and higher (default mode, all versions)** | a:=12.50<br>b:=!2013-12-31!<br>Current date |
| **4D v14 or 4D v15 (preference checked, US version, for instance)** | a:=12.50<br>b:=!12/31/2013!<br>Current date |
| **4D v14 or 4D v15 (preference checked, French version)** | a:=12,50<br>b:=!31/12/2013!<br>Date du jour |

**Note:** When the preference is checked, real and date formats are based on system settings.

## Data Types

In the language, the various types of data that can be stored in a 4D database are referred to as data types. There are eight basic data types: string, numeric, date, time, Boolean, picture, and pointer.

- **String**: A series of characters, such as “Hello there”. Text fields and variables, as well as Alpha fields, are of the String data type
- **Numeric**: Numbers, such as 2 or 1,000.67. Integer, Long Integer, and Real fields and variables are of the numeric data type.
- **Date**: Calendar dates, such as 1/20/89. Date fields and variables are of the date data type.
- **Time**: Times, including hours, minutes, and seconds, such as 1:00:00 or 4:35:30 PM. Time fields and variables are of the time data type.
- **Boolean**: Logical values of TRUE or FALSE. Boolean fields and variables are of the Boolean data type.
- **Picture**: Picture fields and variables are of the picture data type.
- **Pointer**: A special type of data used in advanced programming. Pointer variables are of the pointer data type. There is no corresponding field type.
- **Object**: Composite data type which can contain any type of data sets in the form of key/value pairs. Object fields and variables are of the Object data type.

Note that in the list of data types, the string and numeric data types are associated with more than one type of field. When data is put into a field, the language automatically converts the data to the correct type for the field. For example, if an integer field is used, its data is automatically treated as numeric. In other words, you need not worry about mixing similar field types when using the language; it will manage them for you.

However, when using the language it is important that you do not mix different data types. In the same way that it makes no sense to store “ABC” in a Date field, it makes no sense to put “ABC” in a variable used for dates. In most cases, 4D is very tolerant and will try to make sense of what you are doing. For example, if you add a number to a date, 4D will assume that you want to add that number of days to the date, but if you try to add a string to a date, 4D will tell you that the operation cannot work.

There are cases in which you need to store data as one type and use it as another type. The language contains a full complement of commands that let you convert from one data type to another. For example, you may need to create a part number that starts with a number and ends with characters such as “abc”. In this case, you might write:

```
[Products]Part Number:=String(Number)+"abc"
```

If *Number* is 17, then *[Products]Part Number* will get the string “17abc”.

The data types are fully defined in the section **Data Types**.

## Operators

When you use the language, it is rare that you will simply want a piece of data. It is more likely that you will want to do something to or with that data. You perform such calculations with operators. Operators, in general, take two pieces of data and perform an operation on them that results in a new piece of data. You are already familiar with many operators. For

example, 1 + 2 uses the addition (or plus sign) operator to add two numbers together, and the result is 3. This table shows some familiar numeric operators:

| Operator | Operation | Example |
|---|---|---|
| + | Addition | 1 + 2 results in 3 |
| – | Subtraction | 3 – 2 results in 1 |
| * | Multiplication | 2 * 3 results in 6 |
| / | Division | 6 / 2 results in 3 |

Numeric operators are just one type of operator available to you. 4D supports many different types of data, such as numbers, text, dates, and pictures, so there are operators that perform operations on these different data types.

The same symbols are often used for different operations, depending on the data type. For example, the plus sign (+) performs different operations with different data:

| Data Type | Operation | Example |
|---|---|---|
| Number | Addition | 1 + 2 adds the numbers and results in 3 |
| String | Concatenation | “Hello ” + “there” concatenates (joins together) the strings and results in “Hello there” |
| Date and Number | Date addition | !1989-01-01! + 20 adds 20 days to the date January 1, 1989, and results in the date January 21, 1989 |

The operators are fully defined in the chapter **Operators** and its subsections.

## Expressions

Simply put, expressions return a value. In fact, when using the 4D language, you use expressions all the time and tend to think of them only in terms of the value they represent. Expressions are also sometimes referred to as formulas.

Expressions are made up of almost all the other parts of the language: commands, operators, variables, and fields. You use expressions to build statements (lines of code), which in turn are used to build methods. The language uses expressions wherever it needs a piece of data.

Expressions rarely “stand alone.” There are only a few places in 4D where an expression can be used by itself:

- Query by Formula dialog box
- Debugger where the value of expressions can be checked
- Apply Formula dialog box
- Quick Report editor as a formula for a column

An expression can simply be a constant, such as the number 4 or the string “Hello.” As the name implies, a constant’s value never changes. It is when operators are introduced that expressions start to get interesting. In preceding sections you have already seen expressions that use operators. For example, 4 + 2 is an expression that uses the addition operator to add two numbers together and return the result 6.

You refer to an expression by the data type it returns. There are eight expression types:

- String expression
- Numeric expression (also referred to as number)
- Date expression
- Time expression
- Boolean expression
- Picture expression
- Pointer expression
- Object expression.

The following table gives examples of each type of expression.

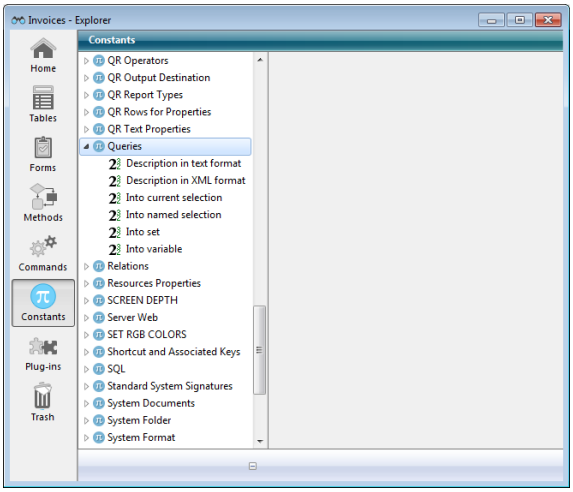
| Expression | Type | Explanation |
|---|---|---|
| “Hello” | String | The word Hello is a string constant, indicated by the double quotation marks. |
| “Hello ” + “there” | String | Two strings, “Hello ” and “there”, are added together (concatenated) with the string concatenation operator (+). The string “Hello there” is returned. |
| “Mr. ” + [People]Name | String | Two strings are concatenated: the string “Mr. ” and the current value of the Name field in the People table. If the field contains “Smith”, the expression returns “Mr. Smith”. |
| **Uppercase**(“smith”) | String | This expression uses **Uppercase**, a command from the language, to convert the string “smith” to uppercase. It returns “SMITH”. |
| 4 | Number | This is a number constant, 4. |
| 4 * 2 | Number | Two numbers, 4 and 2, are multiplied using the multiplication operator (*). The result is the number 8. |
| My Button | Number | This is the name of a button. It returns the current value of the button: 1 if it was clicked, 0 if not. |
| !1997-01-25! | Date | This is a date constant for the date 1/25/97 (January 25, 1997). |
| **Current date** + 30 | Date | This is a date expression that uses the **Current date** command to get today’s date. It adds 30 days to today’s date and returns the new date. |
| ?8:05:30? | Time | This is a time constant that represents 8 hours, 5 minutes, and 30 seconds. |
| ?2:03:04? + ?1:02:03? | Time | This expression adds two times together and returns the time 3:05:07. |
| **True** | Boolean | This command returns the Boolean value TRUE. |
| 10 # 20 | Boolean | This is a logical comparison between two numbers. The number sign (#) means “is not equal to”. Since 10 “is not equal to” 20, the expression returns TRUE. |
| “ABC” = “XYZ” | Boolean | This is a logical comparison between two strings. They are not equal, so the expression returns FALSE. |
| My Picture + 50 | Picture | This expression takes the picture in My Picture, moves it 50 pixels to the right, and returns the resulting picture. |
| ->[People]Name | Pointer | This expression returns a pointer to the field called [People]Name. |
| Table (1) | Pointer | This is a command that returns a pointer to the first table. |
| **JSON Parse**(MyString) | Object | This is a command that returns MyString as an object (if proper format) |

# Constants

A constant is an expression that has a fixed value. There are two types of constants: **predefined constants** that you select by name, and **literal constants** for which you type the actual value.

## Predefined Constants

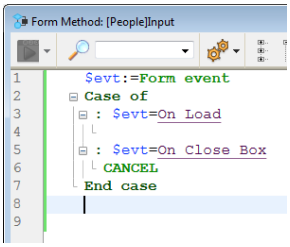
4D provides a set of **predefined constants.** These constants are grouped by themes in the Explorer Window:




To use a predefined constant in a **Method editor** window:

- Drag and drop the constant from the Explorer window to the Method editor window.
- Directly type its name in the Method editor window. The autocomplete mechanism suggests constants that correspond to the programming context.

Predefined constants appeared underlined by default within the Method Editor and Debugger windows:




In the window shown here, On Load, for example, is a predefined constant.

## Literal Constants

Literal Constants can be of four data types:

- String
- Numeric
- Date
- Time

### String Constants

A string constant is enclosed in double, straight quotation marks ("…"). Here are some examples of string constants:

*"Add Records"*
*"No records found."*
*"Invoice"*

An empty string is specified by two quotation marks with nothing between them ("").

### Numeric Constants

A numeric constant is written as a real number. Here are some examples of numeric constants:

*27*
*123.76*
*0.0076*

Negative numbers are specified with the minus sign(−). For example:

*−27*
*−123.76*
*−0.0076*

**Note:** Since 4D v15, the default decimal separator is a period (.), regardless of the system language. If you have checked the "Use regional system settings" option (see **Methods Page**), you must use the separator defined in your system.

### Date Constants

A date constant is enclosed by exclamation marks (!···!). Since 4D v15, a date must be structured using the ISO format (!YYYY-MM-DD!). Here are some examples of date constants:

*!1976-01-01!*
*!2004-09-29!*
*!2015-12-31!*

A null date is specified by *!00-00-00!*.

**Tip**: The Method Editor includes a shortcut for entering a null date. To type a null date, enter the exclamation (!) character and press Enter.

**Notes**:

- For compatibility reasons, 4D accepts two-digit years to be entered. A two-digit year is assumed to be in the 20th or 21st century based on whether it is greater or less than 30, unless this default setting has been changed using the **SET DEFAULT CENTURY** command.
- If you have checked the "Use regional system settings" option (see **Methods Page**), you must use the date format defined in your system. Generally, in a US environment, dates are entered in the form month/day/year, with a slash "/" separating the values.

### Time Constants

A time constant is enclosed by question marks (?...?).

In the US English version of 4D, a time constant is ordered hour:minute:second, with a colon (:) setting off each part. Times are specified in 24-hour format.

Here are some examples of time constants:

*?00:00:00? ` midnight*
*?09:30:00? ` 9:30 am*
*?13:01:59? ` 1 pm, 1 minute, and 59 seconds*

A null time is specified by ?00:00:00?

**Tip**: The Method Editor includes a shortcut for entering a null time. To type a null time, enter the question mark (?) character and press Enter.

# Variables

Data in 4D is stored in two fundamentally different ways. Fields store data permanently on disk; variables store data temporarily in memory.

When you set up your 4D database, you specify the names and types of fields that you want to use. Variables are much the same—you also give them names and different types.

The following variable types correspond to each of the data types:

- String(*) or Text: Alphanumeric string of up to 2 GB of text
- Integer: Integer from -32768 to 32767
- Long Integer: Integer from -2^31 to (2^31)-1
- Real: A number to ±1.7e±308 (13 significant digits)
- Date: 1/1/100 to 12/31/32767
- Time: 00:00:00 to 596000:00:00 (seconds from midnight)
- Boolean: True or False
- Picture: Any Windows or Macintosh picture
- Object: A set of "property/value" pairs structured in a JSON type format
- BLOB (Binary Large OBject): Series of bytes up to 2 GB in size
- Pointer: A pointer to a table, field, variable, array, or array element

(*) In Unicode mode, String and Text type variables are identical. In non-Unicode mode (compatibility mode), a String is a fixed alphanumeric string of up to 255 characters.

You can display variables (except Pointer and BLOB) on the screen, enter data into them, and print them in reports. In these ways, enterable and non-enterable area variables act just like fields, and the same built-in controls are available when you create them:

- Display formats
- Data validation, such entry filters and default values
- Character filters
- Choice lists (hierarchical lists)
- Enterable or non-enterable values

Variables can also do the following:

- Control buttons (buttons, check boxes, radio buttons, 3D buttons, and so on)
- Control sliders (meters, rulers, and dials)
- Control scrollable areas, pop-up menus, and drop-down list boxes
- Control hierarchical lists and hierarchical pop-up menus
- Control button grids, tab controls, picture buttons, and so on
- Display results of calculations that do not need to be saved.

## Creating Variables

You can create variables simply by using them; you do not necessarily need to formally define them as you do with fields. For example, if you want a variable that will hold the current date plus 30 days, you write:

```
MyDate:=Current date+30
```

4D creates *MyDate* and holds the date you need. The line of code reads “MyDate gets the current date plus 30 days.” You could now use *MyDate* wherever you need it in your database. For example, you might need to store the date variable in a field of same type:

```
[MyTable]MyField:=MyDate
```

However, it is usually recommended for a variable to be explicitly defined as a certain type. For more information about typing variables for a database, see the chapter **Compiler**.

## Assigning Data to Variables

Data can be put into and copied out of variables. Putting data into a variable is called **assigning the data to the variable** and is done with the *assignment operator* (*:=*). The assignment operator is also used to assign data to fields.

The assignment operator is the primary way to create a variable and to put data into it. You write the name of the variable that you want to create on the left side of the assignment operator. For example:

```
MyNumber:=3
```

creates the variable **MyNumber** and puts the number *3* into it. If **MyNumber** already exists, then the number *3* is just put into it.

Of course, variables would not be very useful if you could not get data out of them. Once again, you use the assignment operator. If you need to put the value of **MyNumber** in a field called *[Products]Size*, you would write **MyNumber** on the right side of the assignment operator:

```
[Products]Size:=MyNumber
```

In this case, *[Products]Size* would be equal to 3. This example is rather simple, but it illustrates the fundamental way that data is transferred from one place to another by using the language.

**Important**: Be careful not to confuse the assignment operator (*:=*) with the comparison operator, equal (=). Assignment and comparison are very different operations. For more information about the comparison operators, see the section **Operators**.

## Local, Process, and Interprocess Variables

You can create three types of variables: **local** variables, **process** variables, and **interprocess** variables. The difference between the three types of variables is their scope, or the objects to which they are available.

### Local variables

A local variable is, as its name implies, local to a method—accessible only within the method in which it was created and not accessible outside of that method. Being local to a method is formally referred to as being “local in scope.” Local variables are used to restrict a variable so that it works only within the method.

You may want to use a local variable to:

- Avoid conflicts with the names of other variables
- Use data temporarily
- Reduce the number of process variables

The name of a local variable always starts with a dollar sign ($) and can contain up to 31 additional characters. If you enter a longer name, 4D truncates it to the appropriate length.

When you are working in a database with many methods and variables, you often find that you need to use a variable only within the method on which you are working. You can create and use a local variable in the method without worrying about whether you have used the same variable name somewhere else.

Frequently, in a database, small pieces of information are needed from the user. The **Request** command can obtain this information. It displays a dialog box with a message prompting the user for a response. When the user enters the response, the command returns the information the user entered. You usually do not need to keep this information in your methods for very long. This is a typical way to use a local variable. Here is an example:

```
$vsID:=Request("Please enter your ID:")
If(OK=1)
   QUERY([People];[People]ID =$vsID)
End if
```

This method simply asks the user to enter an ID. It puts the response into a local variable, *$vsID*, and then searches for the *ID* that the user entered. When this method finishes, the *$vsID* local variable is erased from memory. This is fine, because the variable is needed only once and only in this method.

### Process variables

A process variable is available only within a process. It is accessible to the process method and any other method called from within the process.

A process variable does not have a prefix before its name. A process variable name can contain up to 31 characters.

In interpreted mode, variables are maintained dynamically; they are created and erased from memory “on the fly.” In compiled mode, all processes you create (user processes) share the same definition of process variables, but each process has a different instance for each variable. For example, the variable *myVar* is one variable in the process *P_1* and another one in the process *P_2*.

A process can “peek and poke” process variables from another process using the commands **GET PROCESS VARIABLE** and **SET PROCESS VARIABLE**. It is good programming practice to restrict the use of these commands to the situation for which they were added to 4D:

- Interprocess communication at specific places or your code
- Handling of interprocess drag and drop
- In Client/Server, communication between processes on client machines and the stored procedures running on the server machines

For more information, see the chapter **Processes** and the description of these commands.

### Interprocess variables

Interprocess variables are available throughout the database and are shared by all processes. They are primarily used to share information between processes.

The name of an interprocess variable always begins with the symbols (<>) — a “less than” sign followed by a “greater than” sign— followed by 31 characters.

**Note**: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

In Client/Server, each machine (Client machines and Server machine) share the same definition of interprocess variables, but each machine has a different instance for each variable.

## Form Object Variables

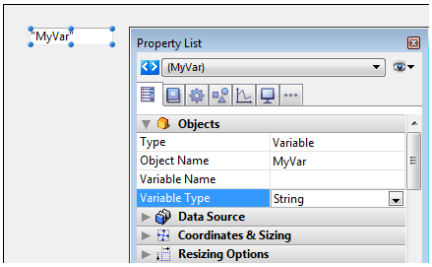
In the Form editor, naming an active object—button, radio button, check box, scrollable area, meter bar, and so on—automatically creates a variable, having the same name by default. For example, if you create a button named *MyButton*, a variable named *MyButton* is also created. Note that this variable name is not the label for the button, but is the name of the button.

The form object variables allow you to control and monitor the objects. For example, when a button is clicked, its variable is set to 1; at all other times, it is 0. The variable associated with a meter or dial lets you read and change the current setting. For example, if you drag a meter to a new setting, the value of the variable changes to reflect the new setting. Similarly, if a method changes the value of the variable, the meter is redrawn to show the new value.

For more information about variables and forms, see the 4D Design Reference Manual as well as the chapter **Form Events**.

### Dynamic variables

You can leave it up to 4D to create variables associated with your form objects (buttons, enterable variables, check boxes, etc.) dynamically and according to your needs. To do this, simply leave the "Variable Name" field blank in the Property list for the object:




When a variable is not named, when the form is loaded, 4D creates a new variable for the object, with a calculated name that is unique in the space of the process variables of the interpreter (which means that this mechanism can be used even in compiled mode). This temporary variable will be destroyed when the form is closed.
In order for this principle to work in compiled mode, it is imperative that dynamic variables are explicitly typed. There are two ways to do this:

- You can set the type using the "Variable Type" menu of the Property list.
  **Note:** When the variable is named, the "Variable Type" menu does not actually type the variable but simply allows the options of the Property list to be updated (except for picture variables). In order to type a named variable, it is necessary to use the commands of the **Compiler** theme.

- You can use a specific initialization code when the form is loaded that uses, for example, the **VARIABLE TO VARIABLE** command:

```
If(Form event=On Load)
   C_TEXT($init)
   $Ptr_object:=OBJECT Get pointer(Object named;"comments")
   $init:=""
   VARIABLE TO VARIABLE(Current process;$Ptr_object->;$init)
End if
```

**Note:** If you specify a dynamic variable, select the value **None** in the "Variable Type" menu, and do not use initialization code, a typing error will be returned by the compiler.

In the 4D code, dynamic variables can be accessed using a pointer obtained with the **OBJECT Get pointer** command. For example:

```
 // assign the time 12:00:00 to the variable for the "tstart" object
$p :=OBJECT Get pointer(Object named;"tstart")
$p->:=?12:00:00?
```

There are two advantages with this mechanism:

- On the one hand, it allows the development of "subform" type components that can be used several times in the same host form. Let us take as an example the case of a *datepicker* subform that is inserted twice in a host form to set a start date and an end date. This subform will use objects for choosing the date of the month and the year. It will be necessary for these objects to work with different variables for the start date and the end date. Letting 4D create their variable with a unique name is a way of resolving this difficulty.
- On the other hand, it can be used to limit memory usage. In fact, form objects only work with process or inter-process variables. However, in compiled mode, an instance of each process variable is created in all the processes, including the server processes. This instance takes up memory, even when the form is not used during the session. Therefore, letting 4D create variables dynamically when loading the forms can economize memory.

**Note:** When there is no variable name, the object name is shown in quotation marks in the form editor (when the object display a variable name by default).

## System Variables

4D maintains a number of variables called **system variables**. These variables let you monitor many operations. System variables are all process variables, accessible only from within a process.

The most important system variable is the **OK** system variable. As its name implies, it tells you if everything is OK in the particular process. Was the record saved? Has the importing operation been completed? Did the user click the OK button? The OK system variable is set to 1 when a task is completed successfully, and to 0 when it is not.

For more information about system variables, see the section **System Variables**.

# System Variables

4D manages **system variables**, which allow you to control the execution of different operations. All system variables are process variables that can only be accessed within one process. This section describes 4D system variables.

For more information about the type of these variables, refer to **System variables** in the **Typing Guide**.

## OK

This is the most commonly used system variable. Usually it is set to 1 when an operation is successfully executed. It is set to 0 when the operation fails. Many 4D commands modify the value of the **OK** system variable. Refer to the description of each command to find out whether it affects this system variable.

In this documentation, the pictogram  indicates that a command modifies the value of the OK variable. You can click on this picture in order to generate a list of all the commands concerned.

## Document

**Document** contains the "long name" (access path+name) of the last file opened or created using the following commands:

| | |
|---|---|
| **Append document** | **BUILD APPLICATION** |
| **Create document** | **_o_Create resource file** |
| **EXPORT DATA** | **EXPORT DIF** |
| **EXPORT SYLK** | **EXPORT TEXT** |
| **GET DOCUMENT ICON** | **IMPORT DATA** |
| **IMPORT DIF** | **IMPORT SYLK** |
| **IMPORT TEXT** | **LOAD SET** |
| **LOAD VARIABLES** | **Open document** |
| **Open resource file** | **PRINT LABEL** |
| **QR REPORT** | **READ PICTURE FILE** |
| **SAVE VARIABLES** | **SAVE SET** |
| **Select document** | **SELECT LOG FILE** |
| **SET CHANNEL** | **USE CHARACTER SET** |
| **WRITE PICTURE FILE** | |

## FldDelimit

**FldDelimit** contains the character code that will be used as a field separator when importing or exporting text. By default, this value is set to 9, which is the character code for the Tab key. To use a different field separator, assign a new value to **FldDelimit**.

## RecDelimit

**RecDelimit** contains the character code that will be used as a record separator when importing or exporting text. By default, this value is set to 13, which is the character code for the Carriage Return key. To use a different record separator, assign a new value to **RecDelimit**.

## Error, Error method, Error line, Error formula

These variables can only be used in an error-catching method installed by the **ON ERR CALL** command. If you want for them to be accessible in the method that caused the error, copy their value into your own process variables.

- **Error**: Longint type system variable. This variable contains the error code. 4D error codes and system error codes are listed in sections of the **Error Codes** theme.
- **Error method**: Text type system variable. This variable contains the full name of the method that triggered the error.
- **Error line**: Longint type system variable. This variable contains the line number at the origin of the error in the method that triggered the error.
- **Error formula**: Text type system variable. This variable contains the formula of the 4D code (raw text) which is at the origin of the error. The text of the formula is expressed in the current language of the 4D code.
  If the source code responsible for the error cannot be found, **Error formula** contains an empty string. This case can occur in compiled databases when:
  - the source code was deleted from the compiled structure using the application builder.
  - the source code is available but the database was compiled without the **Range Checking** option.

## MouseDown, MouseX, MouseY, KeyCode, Modifiers and MouseProc

These system variables can only be used in a method installed by the **ON EVENT CALL** command (except **MouseX** and **MouseY** in some cases, see below).

- **MouseDown** is set to 1 when the mouse button is pushed. Otherwise, it is set to 0.
- If the event is a **MouseDown** (**MouseDown**=1), the **MouseX** and **MouseY** system variables are respectively set to the vertical and horizontal coordinates of the location where the click took place. Both values are expressed in pixels and use the local coordinate system of the window.
- In case of picture fields or variables, the **MouseX** and **MouseY** system variables return the local coordinates of a mouse click in the On Clicked, On Double Clicked and On Mouse Up form events. Local coordinates of the mouse cursor are also returned in the On Mouse Enter and On Mouse Move form events. The coordinates are expressed in pixels with respect to the top left corner of the picture (0,0). For more information, please refer to the **Pictures** section and the **SVG Find element ID by coordinates** command.

- **KeyCode** is set to the character code of the key that was just pressed. If the key is a function key, **KeyCode** is set to a special code. Character codes and function key codes are listed in the sections **Unicode Codes**, **EXPORT TEXT** and **Function Key Codes**.
- **Modifiers** is set to the keyboard modifier keys (**Ctrl/Command**, **Alt/Option**, **Shift**, **Caps Lock**). This variable is only significant in an "interruption on event" installed by the command **ON EVENT CALL**.
- **MouseProc** is set to the process number in which the last event took place.

# Pointers

## Description

Pointers provide an advanced way (in programming) to refer to data.

When you use the language, you access various objects—in particular, tables, fields, variables, and arrays—by simply using their names. However, it is often useful to refer to these elements and access them without knowing their names. This is what pointers let you do.

The concept behind pointers is not that uncommon in everyday life. You often refer to something without knowing its exact identity. For example, you might say to a friend, “Let’s go for a ride in your car” instead of “Let’s go for a ride in the car with license plate 123ABD.” In this case, you are referencing the car with license plate 123ABD by using the phrase “your car.” The phrase “car with license plate 123ABD” is like the name of an object, and using the phrase “your car” is like using a pointer to reference the object.

Being able to refer to something without knowing its exact identity is very useful. In fact, your friend could get a new car, and the phrase “your car” would still be accurate—it would still be a car and you could still take a ride in it. Pointers work the same way. For example, a pointer could at one time refer to a numeric field called Age, and later refer to a numeric variable called Old Age. In both cases, the pointer references numeric data that could be used in a calculation.

You can use pointers to reference tables, fields, variables, arrays, and array elements. The following table gives an example of each data type:

| Object | To Reference | To Use | To Assign |
|---|---|---|---|
| Table | vpTable:=->[Table] | DEFAULT TABLE(vpTable->) | n/a |
| Field | vpField:=->[Table]Field | ALERT(vpField->) | vpField->:="John" |
| Variable | vpVar:=->Variable | ALERT(vpVar->) | vpVar->:="John" |
| Array | vpArr:=->Array | SORT ARRAY(vpArr->;>) | COPY ARRAY (Arr;vpArr->) |
| Array element | vpElem:=->Array{1} | ALERT (vpElem->) | vpElem->:="John" |

## Using Pointers: An Example

It is easiest to explain the use of pointers through an example. This example shows how to access a variable through a pointer. We start by creating a variable:

```
MyVar:="Hello"
```

**MyVar** is now a variable containing the string *“Hello.”* We can now create a pointer to **MyVar**:

```
MyPointer:=->MyVar
```

The -> symbol means “get a pointer to.” This symbol is formed by a dash followed by a “greater than” sign. In this case, it gets the pointer that references or “points to” **MyVar**. This pointer is assigned to **MyPointer** with the assignment operator.

**MyPointer** is now a variable that contains a pointer to **MyVar**. **MyPointer** does not contain *“Hello”*, which is the value in **MyVar**, but you can use **MyPointer** to get this value. The following expression returns the value in **MyVar**:

```
MyPointer->
```

In this case, it returns the string *“Hello”*. The -> symbol, when it follows a pointer, references the object pointed to. This is called **dereferencing**.

It is important to understand that you can use a pointer followed by the -> symbol anywhere that you could have used the object that the pointer points to. This means that you could use the expression **MyPointer->** anywhere that you could use the original **MyVar** variable.

For example, the following line displays an alert box with the word Hello in it:

```
ALERT(MyPointer->)
```

You can also use **MyPointer** to change the data in **MyVar**. For example, the following statement stores the string *"Goodbye"* in the variable **MyVar**:

```
MyPointer->:="Goodbye"
```

If you examine the two uses of the expression **MyPointer->**, you will see that it acts just as if you had used **MyVar** instead. In summary, the following two lines perform the same action—both display an alert box containing the current value in the variable **MyVar**:

```
ALERT(MyPointer->)
ALERT(MyVar)
```

The following two lines perform the same action— both assign the string *"Goodbye"* to **MyVar**:

```
MyPointer->:="Goodbye"
MyVar:="Goodbye"
```

## Using Pointers to Buttons

This section describes how to use a pointer to reference a button. A button is (from the language point of view) nothing more than a variable. Although the examples in this section use pointers to reference buttons, the concepts presented here apply to the use of all types of objects that can be referenced by a pointer.

Let's say that you have a number of buttons in your forms that need to be enabled or disabled. Each button has a condition associated with it that is TRUE or FALSE. The condition says whether to disable or enable the button. You could use a test like this each time you need to enable or disable the button:

```
If(Condition) ` If the condition is TRUE…
   OBJECT SET ENABLED(MyButton;True) ` enable the button
Else ` Otherwise…
   OBJECT SET ENABLED(MyButton;False) ` disable the button
End if
```

You would need to use a similar test for every button you set, with only the name of the button changing. To be more efficient, you could use a pointer to reference each button and then use a subroutine for the test itself.

You must use pointers if you use a subroutine, because you cannot refer to the button's variables in any other way. For example, here is a project method called **SET BUTTON**, which references a button with a pointer:

```
  ` SET BUTTON project method
  ` SET BUTTON ( Pointer ; Boolean )
  ` SET BUTTON ( -> Button ; Enable or Disable )
  `
  ` $1 - Pointer to a button
  ` $2 - Boolean. If TRUE, enable the button. If FALSE, disable the button

If($2) ` If the condition is TRUE…
   OBJECT SET ENABLED($1->;True) ` enable the button
Else ` Otherwise…
   OBJECT SET ENABLED($1->;False) ` disable the button
End if
```

You can call the **SET BUTTON** project method as follows:

```
  ` ...
SET BUTTON(->bValidate;True)
  ` ...
SET BUTTON(->bValidate;False)
  ` ...
SET BUTTON(->bValidate;([Employee]Last Name#""))
  ` ...
For($vlRadioButton;1;20)
   $vpRadioButton:=Get pointer("r"+String($vlRadioButton))
   SET BUTTON($vpRadioButton;False)
End for
```

## Using Pointers to Tables

Anywhere that the language expects to see a table, you can use a dereferenced pointer to the table.
You create a pointer to a table by using a line like this:

```
TablePtr:=->[anyTable]
```

You can also get a pointer to a table by using the **Table** command. For example:

```
TablePtr:=Table(20)
```

You can use the dereferenced pointer in commands, like this:

```
DEFAULT TABLE(TablePtr->)
```

## Using Pointers to Fields

Anywhere that the language expects to see a field, you can use a dereferenced pointer to reference the field. You create a pointer to a field by using a line like this:

```
FieldPtr:=->[aTable]ThisField
```

You can also get a pointer to a field by using the **Field** command. For example:

```
FieldPtr:=Field(1;2)
```

You can use the dereferenced pointer in commands, like this:

```
OBJECT SET FONT(FieldPtr->;"Arial")
```

## Using Pointers to Variables

The example at the beginning of this section illustrates the use of a pointer to a variable:

```
MyVar:="Hello"
MyPointer:=->MyVar
```

You can use pointers to interprocess, process and, starting with version 2004.1, local variables.

When you use pointers to process or local variables, you must be sure that the variable pointed to is already set when the pointer is used. Keep in mind that local variables are deleted when the method that created them has completed its execution and process variables are deleted at the end of the process that created them. When a pointer calls a variable that no longer exists, this causes a syntax error in interpreted mode (variable not defined) but it can generate a more serious error in compiled mode.

**Note about local variables:** Pointers to local variables allow you to save process variables in many cases. Pointers to local variables can only be used within the same process.
In the debugger, when you display a pointer to a local variable that has been declared in another method, the original method name is indicated in parentheses, after the pointer. For example, if you write in Method1:

```
$MyVar:="Hello world"
Method2(->$MyVar)
```

In Method2, the debugger will display *$1* as follows:

```
$1   ->$MyVar (Method1)
```

The value of *$1* will be:

```
$MyVar (Method1)   "Hello world"
```

## Using Pointers to Array Elements

You can create a pointer to an array element. For example, the following lines create an array and assign a pointer to the first array element to a variable called **ElemPtr**:

```
ARRAY REAL(anArray;10) ` Create an array
ElemPtr:=->anArray{1} ` Create a pointer to the array element
```

You could use the dereferenced pointer to assign a value to the element, like this:

```
ElemPtr->:=8
```

## Using Pointers to Arrays

You can create a pointer to an array. For example, the following lines create an array and assign a pointer to the array to a variable called **ArrPtr**:

```
ARRAY REAL(anArray;10) ` Create an array
ArrPtr:=->anArray ` Create a pointer to the array
```

It is important to understand that the pointer points to the array; it does not point to an element of the array. For example, you can use the dereferenced pointer from the preceding lines like this:

```
SORT ARRAY(ArrPtr->;>) ` Sort the array
```

If you need to refer to the fourth element in the array by using the pointer, you do this:

```
ArrPtr->{4}:=84
```

## Using an Array of Pointers

It is often useful to have an array of pointers that reference a group of related objects.

One example of such a group of objects is a grid of variables in a form. Each variable in the grid is sequentially numbered, for example: **Var1**,**Var2**,…, **Var10**. You often need to reference these variables indirectly with a number. If you create an array of pointers, and initialize the pointers to point to each variable, you can then easily reference the variables. For example, to create an array and initialize each element, you could use the following lines:

```
ARRAY POINTER(apPointers;10) ` Create an array to hold 10 pointers
For($i;1;10) ` Loop once for each variable
   apPointers{$i}:=Get pointer("Var"+String($i)) ` Initialize the array element
End for
```

The **Get pointer** function returns a pointer to the named object.

To reference any of the variables, you use the array elements. For example, to fill the variables with the next ten dates (assuming they are variables of the date type), you could use the following lines:

```
For($i;1;10) ` Loop once for each variable
   apPointers{$i}->:=Current date+$i  ` Assign the dates
End for
```

## Setting a Button Using a Pointer

If you have a group of related radio buttons in a form, you often need to set them quickly. It is inefficient to directly reference each one of them by name. Let's say you have a group of radio buttons named **Button1**, **Button2**,…, **Button5**.

In a group of radio buttons, only one radio button is on. The number of the radio button that is on can be stored in a numeric field. For example, if the field called *[Preferences]Setting* contains *3*, then **Button3** is selected. In your form method, you could use the following code to set the button:

```
Case of
   :(Form event=On Load)
` ...
      Case of
         :([Preferences]Setting=1)
```

```
            Button1:=1
         :([Preferences]Setting=2)
            Button2:=1
         :([Preferences]Setting=3)
            Button3:=1
         :([Preferences]Setting=4)
            Button4:=1
         :([Preferences]Setting=5)
            Button5:=1
      End case
   ` ...
End case
```

A separate case must be tested for each radio button. This could be a very long method if you have many radio buttons in your form. Fortunately, you can use pointers to solve this problem. You can use the **Get pointer** command to return a pointer to a radio button. The following example uses such a pointer to reference the radio button that must be set. Here is the improved code:

```
Case of
   :(Form event=On Load)
   ` ...
      $vpRadio:=Get pointer("Button"+String([Preferences]Setting))
      $vpRadio->:=1
   ` ...
End case
```

The number of the set radio button must be stored in the field called *[Preferences]Setting*. You can do so in the form method for the On Clicked event:

```
[Preferences]Setting:=Button1+(Button2*2)+(Button3*3)+(Button4*4)+(Button5*5)
```

## Passing Pointers to Methods

You can pass a pointer as a parameter to a method. Inside the method, you can modify the object referenced by the pointer. For example, the following method, **TAKE TWO**, takes two parameters that are pointers. It changes the object referenced by the first parameter to uppercase characters, and the object referenced by the second parameter to lowercase characters. Here is the method:

```
   ` TAKE TWO project method
   ` $1 - Pointer to a string field or variable. Change this to uppercase.
   ` $2 - Pointer to a string field or variable. Change this to lowercase.
$1->:=Uppercase($1->)
$2->:=Lowercase($2->)
```

The following line uses the **TAKE TWO** method to change a field to uppercase characters and to change a variable to lowercase characters:

```
TAKE TWO(->[My Table]My Field;->MyVar)
```

If the field *[My Table]My Field* contained the string *"jones",* it would be changed to the string *"JONES"*. If the variable **MyVar** contained the string *"HELLO",* it would be changed to the string *"hello"*.

In the **TAKE TWO** method, and in fact, whenever you use pointers, it is important that the data type of the object being referenced is correct. In the previous example, the pointers must point to an object that contains a string or text.

## Pointers to Pointers

If you really like to complicate things, you can use pointers to reference other pointers. Consider this example:

```
MyVar:="Hello"
PointerOne:=->MyVar
PointerTwo:=->PointerOne
(PointerTwo->)->:="Goodbye"
ALERT((Point Two->)->)
```

It displays an alert box with the word *"Goodbye"* in it.

Here is an explanation of each line of the example:

- *MyVar:="Hello"*
  --> This line puts the string *"Hello"* into the variable **MyVar**.
- *PointerOne:=->MyVar*
  --> **PointerOne** now contains a pointer to **MyVar**.
- *PointerTwo:=->PointerOne*
  --> **PointerTwo** (a new variable) contains a pointer to **PointerOne**, which in turn points to **MyVar**.
- *(PointerTwo->)->:="Goodbye"*
  --> **PointerTwo->** references the contents of **PointerOne**, which in turn references **MyVar**. Therefore (**PointerTwo->)->** references the contents of **MyVar**. So in this case, **MyVar** is assigned *"Goodbye"*.
- *ALERT ((PointerTwo->)->)*
  --> Same thing: **PointerTwo->** references the contents of **PointerOne**, which in turn references **MyVar**. Therefore (**PointerTwo->)->** references the contents of **MyVar**. So in this case, the alert box displays the contents of *myVar*.

The following line puts *"Hello"* into **MyVar**:

```
(PointerTwo->)->:="Hello"
```

The following line gets *"Hello"* from **MyVar** and puts it into **NewVar**:

```
NewVar:=(PointerTwo->)->
```

**Important**: Multiple dereferencing requires parentheses.

# Identifiers

This section describes the conventions for naming various objects in the 4D language. The names for all objects follow these rules:

- A name must begin with an alphabetic character or an underscore.
- Thereafter, the name can include alphabetic characters, numeric characters, the space character, and the underscore character.
- Periods, slashes, quotation marks and colons are not allowed.
- Characters reserved for use as operators, such as * and +, are not allowed.
- 4D ignores any trailing spaces.

**Note**: Additional rules need to be respected when objects need to be handled via SQL: only the characters *_0123456789abcdefghijklmnopqrstuvwxyz* are accepted, and the name must not include any SQL keywords (command, attribute, etc.). The "SQL" area of the Inspector in the Structure editor automatically indicates any unauthorized characters in the name of a table or field.

## Tables

You denote a table by placing its name between brackets: *[...]*. A table name can contain up to 31 characters.

Examples

```
DEFAULT TABLE([Orders])
FORM SET INPUT([Clients];"Entry")
ADD RECORD([Letters])
```

## Fields

You denote a field by first specifying the table to which the field belongs. The field name immediately follows the table name. A field name can contain up to 31 characters.

Examples

```
[Orders]Total:=Sum([Line]Amount)
QUERY([Clients];[Clients]Name="Smith")
[Letters]Text:=Capitalize text([Letters]Text)
```

## Interprocess Variables

You denote an interprocess variable by preceding the name of the variable with the symbols (<>) — a “less than” sign followed by a “greater than” sign.

**Note**: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

An interprocess variable can have up to 31 characters, not including the <> symbols.

Examples

```
<>vlProcessID:=Current process
<>vsKey:=Char(KeyCode)
If(<>vtName#"")
```

## Process Variables

You denote a process variable by using its name (which cannot start with the <> symbols nor the dollar sign *$*). A process variable name can contain up to 31 characters.

Examples

```
<>vrGrandTotal:=Sum([Accounts]Amount)
If(bValidate=1)
   vsCurrentName:=""
```

## Local Variables

You denote a local variable with a dollar sign ($) followed by its name. A local variable name can contain up to 31 characters, not including the dollar sign.

Examples

```
For($vlRecord;1;100)
   If($vsTempVar="No")
      $vsMyString:="Hello there"
```

## Arrays

You denote an array by using its name, which is the name you passed to the array declaration (such as **ARRAY LONGINT**) when you created the array. Arrays are variables, and from the scope point of view, like variables, there are three different types of arrays:

- Interprocess arrays,
- Process arrays,
- Local arrays.

**Interprocess Arrays**

The name of an interprocess array is preceded by the symbols (<>) — a “less than” sign followed by a “greater than” sign.

**Note**: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

An interprocess array name can contain up to 31 characters, not including the <> symbols.

Examples

```
ARRAY TEXT(<>atSubjects;Records in table([Topics]))
SORT ARRAY(<>asKeywords;>)
ARRAY INTEGER(<>aiBigArray;10000)
```

**Process Arrays**

You denote a process array by using its name (which cannot start with the <> symbols nor the dollar sign $). A process array name can contain up to 31 characters.

Examples

```
ARRAY TEXT(atSubjects;Records in table([Topics]))
SORT ARRAY(asKeywords;>)
ARRAY INTEGER(aiBigArray;10000)
```

**Local Arrays**

The name of a local array is preceded by the dollar sign ($). An local array name can contain up to 31 characters, not including the dollar sign.

Examples

```
ARRAY TEXT($atSubjects;Records in table([Topics]))
SORT ARRAY($asKeywords;>)
ARRAY INTEGER($aiBigArray;10000)
```

**Elements of arrays**

You reference an element of an interprocess, process or local array by using the curly braces({…}). The element referenced is denoted by a numeric expression.

Examples

```
  ` Addressing an element of an interprocess array
If(<>asKeywords{1}="Stop")
   <>atSubjects{$vlElem}:=[Topics]Subject
   $viNextValue:=<>aiBigArray{Size of array(<>aiBigArray)}

  ` Addressing an element of a process array
   If(asKeywords{1}="Stop")
      atSubjects{$vlElem}:=[Topics]Subject
      $viNextValue:=aiBigArray{Size of array(aiBigArray)}

  ` Addressing an element of a local array
      If($asKeywords{1}="Stop")
         $atSubjects{$vlElem}:=[Topics]Subject
         $viNextValue:=$aiBigArray{Size of array($aiBigArray)}
```

**Elements of two-dimensional arrays**

You reference an element of a two-dimensional array by using the curly braces ({…}) twice. The element referenced is denoted by two numeric expressions in two sets of curly braces.

Examples

```
  ` Addressing an element of a two-dimensional interprocess array
If(<>asKeywords{$vlNextRow}{1}="Stop")
   <>atSubjects{10}{$vlElem}:=[Topics]Subject
   $viNextValue:=<>aiBigArray{$vlSet}{Size of array(<>aiBigArray{$vlSet})}

  ` Addressing an element of a two-dimensional process array
   If(asKeywords{$vlNextRow}{1}="Stop")
      atSubjects{10}{$vlElem}:=[Topics]Subject
      $viNextValue:=aiBigArray{$vlSet}{Size of array(aiBigArray{$vlSet})}

  ` Addressing an element of a two-dimensional local array
      If($asKeywords{$vlNextRow}{1}="Stop")
         $atSubjects{10}{$vlElem}:=[Topics]Subject
         $viNextValue:=$aiBigArray{$vlSet}{Size of array($aiBigArray{$vlSet})}
```

## Forms

You denote a form by using a string expression that represents its name. A form name can contain up to 31 characters.

Examples

```
FORM SET INPUT([People];"Input")
FORM SET OUTPUT([People];"Output")
DIALOG([Storage];"Note box"+String($vlStage))
```

## Form objects

You designate a form object by passing its name as a string, preceded by the* parameter. An object name can contain up to 255 bytes.

Example :

```
OBJECT SET FONT(*;"Binfo";"Times")
```

See also the **Object Properties** section.

## Methods

You denote a method (procedure and function) by using its name. A method name can contain up to 31 characters.

**Note**: A method that does not return a result is also called a **procedure**. A method that returns a result is also called a **function**.

Examples

```
If(New client)
   DELETE DUPLICATED VALUES
   APPLY TO SELECTION([Employees];INCREASE SALARIES)
```

**Tip**: It is a good programming technique to adopt the same naming convention as the one used by 4D for built-in commands. Use uppercase characters for naming your methods; however if a method is a function, capitalize the first character of its name. By doing so, when you reopen a database for maintenance after a few months, you will already know if a method returns a result by simply looking at its name in the Explorer window.

**Note**: When you call a method, you just type its name. However, some 4D built-in commands, such as **ON EVENT CALL**, as well as all the Plug-In commands, expect the name of a method as a string when a method parameter is passed. Example:

Examples

```
 ` This command expects a method (function) or formula
QUERY BY FORMULA([aTable];Special query)
 ` This command expects a method (procedure) or statement
APPLY TO SELECTION([Employees];INCREASE SALARIES)
 ` But this command expects a method name
ON EVENT CALL("HANDLE EVENTS")
 ` And this Plug-In command expects a method name
WR ON ERROR("WR HANDLE ERRORS")
```

Methods can accept parameters (arguments). The parameters are passed to the method in parentheses, following the name of the method. Each parameter is separated from the next by a semicolon (*;*). The parameters are available within the called method as consecutively numbered local variables: *$1*, *$2*,…, *$n*. In addition, multiple consecutive (and last) parameters can be addressed with the syntax *${n}*where n, numeric expression, is the number of the parameter.

Inside a function, the *$0* local variable contains the value to be returned.

Examples

```
 ` Within DROP SPACES $1 is a pointer to the field [People]Name
DROP SPACES(->[People]Name)

 ` Within Calc creator:
 ` - $1 is numeric and equal to 1
 ` - $2 is numeric and equal to 5
 ` - $3 is text or string and equal to "Nice"
 ` - The result value is assigned to $0
$vsResult:=Calc creator(1;5;"Nice")

 ` Within Dump:
 ` - The three parameters are text or string
 ` - They can be addressed as $1, $2 or $3
 ` - They can also be addressed as, for instance, ${$vlParam} where $vlParam is 1, 2 or 3
 ` - The result value is assigned to $0
vtClone:=Dump("is";"the";"it")
```

## Plug-In Commands (External Procedures, Functions and Areas)

You denote a plug-in command by using its name as defined by the plug-in. A plug-in command name can contain up to 31 characters.

Examples

```
WR BACKSPACE(wrArea;0)
$pvNewArea:=PV New offscreen area
```

## Sets

From the scope point of view, there are two types of sets:

- Interprocess sets
- Process sets.

4D Server also includes:

- Client sets.

**Interprocess Sets**
A set is an interprocess set if the name of the set is preceded by the symbols (<>) — a “less than” sign followed by a “greater than” sign.

**Note**: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

An interprocess set name can contain up to 255 characters, not including the <> symbols.

**Process Sets**
You denote a process set by using a string expression that represents its name (which cannot start with the <> symbols or the dollar sign *$*). A set name can contain up to 255 characters.

**Client Sets**
The name of a client set is preceded by the dollar sign (*$*). A client set name can contain up to 255 characters, not including the dollar sign.

**Note**: Sets are maintained on the Server machine. In certain cases, for efficiency or special purposes, you may need to work with sets locally on the Client machine. To do so, you use Client sets.

Examples

```
  ` Interprocess sets
USE SET("<>Deleted Records")
CREATE SET([Customers];"<>Customer Orders")
If(Records in set("<>Selection"+String($i))>0)
  ` Process sets
   USE SET("Deleted Records")
   CREATE SET([Customers];"Customer Orders")
   If(Records in set("<>Selection"+String($i))>0)
  ` Client sets
      USE SET("$Deleted Records")
      CREATE SET([Customers];"$Customer Orders")
      If(Records in set("$Selection"+String($i))>0)
```

## Named Selections

From the scope point of view, there are two types of named selections:

- Interprocess named selections
- Process named selections.

**Interprocess Named Selections**
A named selection is an interprocess named selection if its name is preceded by the symbols (<>) — a “less than” sign followed by a “greater than” sign.

**Note**: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

An interprocess named selection name can contain up to 255 characters, not including the <> symbols.

**Process Named Selections**
You denote a process named selection by using a string expression that represents its name (which cannot start with the <> symbols nor the dollar sign *$*). A named selection name can contain up to 255 characters.

Examples

```
  ` Interprocess Named Selection
USE NAMED SELECTION([Customers];"<>ByZipcode")
  ` Process Named Selection
USE NAMED SELECTION([Customers];"<>ByZipcode")
```

## Processes

In the single-user version, or in Client/Server on the Client side, there are two types of processes:

- Global processes
- Local processes.

**Global Processes**

You denote a global process by using a string expression that represents its name (which cannot start with the dollar sign *$*). A process name can contain up to 255 characters.

**Local Processes**

You denote a local process if the name of the process is preceded by a dollar (*$*) sign. The process name can contain up to 255 characters, not including the dollar sign.

Example

```
  ` Starting the global process "Add Customers"
$vlProcessID:=New process("P_ADD_CUSTOMERS";48*1024;"Add Customers")
  ` Starting the local process "$Follow Mouse Moves"
$vlProcessID:=New process("P_MOUSE_SNIFFER";16*1024;"$Follow Mouse Moves")
```

## Summary of Naming Conventions

The following table summarizes 4D naming conventions.

| Type | Max. Length | Example |
|---|---|---|
| Table | 31 | [Invoices] |
| Field | 31 | [Employees]Last Name |
| Interprocess Variable | <> + 31 | <>vlNextProcessID |
| Process Variable | 31 | vsCurrentName |
| Local Variable | $ + 31 | $vlLocalCounter |
| Form | 31 | "My Custom Web Input" |
| Form object | 31 | "MyButton" |
| Interprocess Array | <> + 31 | <>apTables |
| Process Array | 31 | asGender |
| Local Array | $ + 31 | $atValues |
| Method | 31 | M_ADD_CUSTOMERS |
| Plug-in Routine | 31 | WR INSERT TEXT |
| Interprocess Set | <> + 255 | "<>Records to be Archived" |
| Process Set | 255 | "Current selected records" |
| Client Set | $ + 255 | "$Previous Subjects" |
| Named Selection | 255 | "Employees A to Z" |
| Interprocess Named Selection | <> + 255 | "<>Employees Z to A" |
| Local Process | $ + 255 | "$Follow Events" |
| Global Process | 255 | "P_INVOICES_MODULE" |
| Semaphore | 255 | "mysemaphore" |

## Resolving Naming Conflicts

If a particular object has the same name as another object of a different type (for example, if a field is named Person and a variable is also named Person), 4D uses a priority system to identify the object. It is up to you to ensure that you use unique names for the parts of your database.

4D identifies names used in procedures in the following order:

1. Fields
2. Commands
3. Methods
4. Plug-in routines
5. Predefined constants
6. Variables.

For example, 4D has a built-in command called **Date**. If you named a method Date, 4D would recognize it as the built-in **Date** command, and not as your method. This would prevent you from calling your method. If, however, you named a field *"Date"*, 4D would try to use your field instead of the **Date** command.

# Control Flow

Regardless of the simplicity or complexity of a method, you will always use one or more of three types of programming structures. Programming structures control the flow of execution, whether and in what order statements are executed within a method. There are three types of structures:

- Sequential
- Branching
- Looping

The 4D language contains statements that control each of these structures.

## Sequential structure

The sequential structure is a simple, linear structure. A sequence is a series of statements that 4D executes one after the other, from first to last. For example:

```
OUTPUT FORM([People];"Listing")
ALL RECORDS([People])
DISPLAY SELECTION([People])
```

A one-line routine, frequently used for object methods, is the simplest case of a sequential structure. For example:

```
[People]Last Name:=Uppercase([People]Last Name)
```

**Note:** The **Begin SQL** / **End SQL** keywords can be used to delimit sequential structures to be executed by the SQL engine of 4D. For more information, please refer to the description of these keywords.

## Branching structures

A branching structure allows methods to test a condition and take alternative paths, depending on the result. The condition is a Boolean expression, an expression that evaluates TRUE or FALSE. One branching structure is the **If...Else...End if** structure, which directs program flow along one of two paths. The other branching structure is the **Case of...Else...End case** structure, which directs program flow to one of many paths.

## Looping structures

When writing methods, it is very common to find that you need a sequence of statements to repeat a number of times. To deal with this need, the language provides three looping structures:

- **While...End while**
- **Repeat...Until**
- **For...End for**

The loops are controlled in two ways: either they loop until a condition is met, or they loop a specified number of times. Each looping structure can be used in either way, but While loops and Repeat loops are more appropriate for repeating until a condition is met, and For loops are more appropriate for looping a specified number of times.

**Note**: 4D allows you to embed programming structures (If/While/For/Case of/Repeat) up to a "depth" of 512 levels.

## If...Else...End if

---

The formal syntax of the **If...Else...End if** control flow structure is:

```
If(Boolean_Expression)
   statement(s)
Else
   statement(s)
End if
```

Note that the **Else** part is optional; you can write:

```
If(Boolean_Expression)
   statement(s)
End if
```

The **If...Else...End if** structure lets your method choose between two actions, depending on whether a test (a Boolean expression) is TRUE or FALSE.

When the Boolean expression is TRUE, the statements immediately following the test are executed. If the Boolean expression is FALSE, the statements following the **Else** statement are executed. The **Else** statement is optional; if you omit **Else**, execution continues with the first statement (if any) following the **End if**.

Note that the Boolean expression is always fully evaluated. Consider in particular the following test:

```
If(MethodA & MethodB)
   ...
End if
```

The expression is TRUE only if both methods are TRUE. However, even if ***MethodA*** returns FALSE, 4D will still evaluate ***MethodB***, which is a useless waste of time. In this case, it is more interesting to use a structure like:

```
If(MethodA)
   If(MethodB)
      ...
   End if
End if
```

The result is similar and ***MethodB*** is evaluated only if necessary.

### Example

```
   ` Ask the user to enter a name
$Find:=Request(Type a name)
If(OK=1)
   QUERY([People];[People]LastName=$Find)
Else
   ALERT("You did not enter a name.")
End if
```

**Tip**: Branching can be performed without statements to be executed in one case or the other. When developing an algorithm or a specialized application, nothing prevents you from writing:

```
If(Boolean_Expression)
Else
   statement(s)
End if
```

or:

```
If(Boolean_Expression)
   statement(s)
Else
End if
```

## Case of...Else...End case

The formal syntax of the **Case of...Else...End case** control flow structure is:

```
Case of
   :(Boolean_Expression)
      statement(s)
   :(Boolean_Expression)
      statement(s)
      .
      .
      .

   :(Boolean_Expression)
      statement(s)
   Else
      statement(s)
End case
```

Note that the **Else** part is optional; you can write:

```
Case of
   :(Boolean_Expression)
      statement(s)
   :(Boolean_Expression)
      statement(s)
      .
      .
      .

   :(Boolean_Expression)
      statement(s)
End case
```

As with the **If...Else...End if** structure, the **Case of...Else...End case** structure also lets your method choose between alternative actions. Unlike the **If...Else...End if** structure, the **Case of...Else...End case** structure can test a reasonable unlimited number of Boolean expressions and take action depending on which one is TRUE.

Each Boolean expression is prefaced by a colon (*:*). This combination of the colon and the Boolean expression is called a case. For example, the following line is a case:

```
:(bValidate=1)
```

Only the statements following the first TRUE case (and up to the next case) will be executed. If none of the cases are TRUE, none of the statements will be executed (if no **Else** part is included).

You can include an **Else** statement after the last case. If all of the cases are FALSE, the statements following the **Else** will be executed.

### Example

This example tests a numeric variable and displays an alert box with a word in it:

```
Case of
   :(vResult=1) ` Test if the number is 1
      ALERT("One.") ` If it is 1, display an alert
   :(vResult=2) ` Test if the number is 2
      ALERT("Two.") ` If it is 2, display an alert
   :(vResult=3) ` Test if the number is 3
      ALERT("Three.") ` If it is 3, display an alert
   Else ` If it is not 1, 2, or 3, display an alert
```

```
      ALERT("It was not one, two, or three.")
End case
```

For comparison, here is the **If...Else...End if** version of the same method:

```
If(vResult=1) ` Test if the number is 1
   ALERT("One.") ` If it is 1, display an alert
Else
   If(vResult=2) ` Test if the number is 2
      ALERT("Two.") ` If it is 2, display an alert
   Else
      If(vResult=3) ` Test if the number is 3
         ALERT("Three.") ` If it is 3, display an alert
      Else ` If it is not 1, 2, or 3, display an alert
         ALERT("It was not one, two, or three.")
      End if
   End if
End if
```

Remember that with a **Case of...Else...End case** structure, only the first TRUE case is executed. Even if two or more cases are TRUE, only the statements following the first TRUE case will be executed.

Consequently, when you want to implement hierarchical tests, you should make sure the condition statements that are lower in the hierarchical scheme appear first in the test sequence. For example, the test for the presence of condition1 covers the test for the presence of condition1&condition2 and should therefore be located last in the test sequence. For example, the following code will never see its last condition detected:

```
Case of
   :(vResult=1)
      ... `statement(s)
   :((vResult=1) & (vCondition#2)) `this case will never be detected
      ... `statement(s)
End case
.
```

In the code above, the presence of the second condition is not detected since the test "vResult=1" branches off the code before any further testing. For the code to operate properly, you can write it as follows:

```
Case of
   :((vResult=1) & (vCondition#2)) `this case will be detected first
      ... `statement(s)
   :(vResult=1)
      ... `statement(s)
End case
.
```

Also, if you want to implement hierarchical testing, you may consider using hierarchical code.

**Tip**: Branching can be performed without statements to be executed in one case or another. When developing an algorithm or a specialized application, nothing prevents you from writing:

```
Case of
   :(Boolean_Expression)
   :(Boolean_Expression)

      .
      .
      .

   :(Boolean_Expression)
      statement(s)
   Else
      statement(s)
End case
```

or:

```
Case of
   :(Boolean_Expression)
   :(Boolean_Expression)
      statement(s)
      .
      .
      .

   :(Boolean_Expression)
      statement(s)
   Else
End case
```

or:

```
Case of
   Else
      statement(s)
End case
```

## While...End while

The formal syntax of the **While...End while** control flow structure is:

```
While(Boolean_Expression)
   statement(s)
End while
```

A **While...End while** loop executes the statements inside the loop as long as the Boolean expression is TRUE. It tests the Boolean expression at the beginning of the loop and does not enter the loop at all if the expression is FALSE.

It is common to initialize the value tested in the Boolean expression immediately before entering the **While...End while** loop. Initializing the value means setting it to something appropriate, usually so that the Boolean expression will be TRUE and **While...End while** executes the loop.

The Boolean expression must be set by something inside the loop or else the loop will continue forever. The following loop continues forever because NeverStop is always TRUE:

```
NeverStop:=True
While(NeverStop)
End while
```

If you find yourself in such a situation, where a method is executing uncontrolled, you can use the trace facilities to stop the loop and track down the problem. For more information about tracing a method, see the chapter **Debugging**.

### Example

```
CONFIRM("Add a new record?") ` The user wants to add a record?
While(OK=1) ` Loop as long as the user wants to
   ADD RECORD([aTable]) ` Add a new record
End while ` The loop always ends with End while
```

In this example, the OK system variable is set by the **CONFIRM** command before the loop starts. If the user clicks the OK button in the confirmation dialog box, the OK system variable is set to 1 and the loop starts. Otherwise, the OK system variable is set to 0 and the loop is skipped. Once the loop starts, the **ADD RECORD** command keeps the loop going because it sets the OK system variable to 1 when the user saves the record. When the user cancels (does not save) the last record, the OK system variable is set to 0 and the loop stops.

# Repeat...Until

The formal syntax of the **Repeat...Until** control flow structure is:

```
Repeat
   statement(s)
Until(Boolean_Expression)
```

A **Repeat...Until** loop is similar to a **While...End while** loop, except that it tests the Boolean expression after the loop rather than before. Thus, a **Repeat...Until** loop always executes the loop once, whereas if the Boolean expression is initially False, a **While...End while** loop does not execute the loop at all.

The other difference with a **Repeat...Until** loop is that the loop continues until the Boolean expression is TRUE.

## Example

Compare the following example with the example for the **While...End while** loop. Note that the Boolean expression does not need to be initialized—there is no **CONFIRM** command to initialize the OK variable.

```
Repeat
   ADD RECORD([aTable])
Until(OK=0)
```

# For...End for

The formal syntax of the **For...End for** control flow structure is:

```
For(Counter_Variable;Start_Expression;End_Expression{;Increment_Expression})
   statement(s)
End for
```

The **For...End for** loop is a loop controlled by a counter variable:

- The counter variable **Counter_Variable** is a numeric variable (Real, Integer, or Long Integer) that the **For...End for** loop initializes to the value specified by **Start_Expression**.
- Each time the loop is executed, the counter variable is incremented by the value specified in the optional value **Increment_Expression**. If you do not specify **Increment_Expression**, the counter variable is incremented by one (*1*), which is the default.
- When the counter variable passes the **End_Expression** value, the loop stops.

**Important**: The numeric expressions **Start_Expression**, **End_Expression** and **Increment_Expression** are evaluated once at the beginning of the loop. If these expressions are variables, changing one of these variables **within** the loop **will not** affect the loop.

**Tip**: However, for special purposes, you can change the value of the counter variable **Counter_Variable** **within** the loop; this **will** affect the loop.

- Usually **Start_Expression** is less than **End_Expression**.
- If **Start_Expression** and **End_Expression** are equal, the loop will execute only once.
- If **Start_Expression** is greater than **End_Expression,** the loop will not execute at all unless you specify a negative **Increment_Expression**. See the examples.

## Basic Examples

1. The following example executes 100 iterations:

```
For(vCounter;1;100)
  ` Do something
End for
```

2. The following example goes through all elements of the array *anArray*:

```
For($vlElem;1;Size of array(anArray))
  ` Do something with the element
   anArray{$vlElem}:=...
End for
```

3. The following example goes through all the characters of the text *vtSomeText*:

```
For($vlChar;1;Length(vtSomeText))
  ` Do something with the character if it is a TAB
   If(Character code(vtSomeText≤$vlChar≥)=Tab)
  ` ...
   End if
End for
```

4. The following example goes through the selected records for the table *[aTable]*:

```
FIRST RECORD([aTable])
For($vlRecord;1;Records in selection([aTable]))
  ` Do something with the record
   SEND RECORD([aTable])
  ` ...
```

```
   ` Go to the next record
   NEXT RECORD([aTable])
End for
```

Most of the **For...End for** loops you will write in your databases will look like the ones listed in these examples.

## Decrementing variable counter

In some cases, you may want to have a loop whose counter variable is decreasing rather than increasing. To do so, you must specify **Start_Expression** greater than **End_Expression** and a negative **Increment_Expression**. The following examples do the same thing as the previous examples, but in reverse order:

5. The following example executes 100 iterations:

```
For(vCounter;100;1;-1)
   ` Do something
End for
```

6. The following example goes through all elements of the array *anArray*:

```
For($vlElem;Size of array(anArray);1;-1)
   ` Do something with the element
   anArray{$vlElem}:=...
End for
```

7. The following example goes through all the characters of the text *vtSomeText*:

```
For($vlChar;Length(vtSomeText);1;-1)
   ` Do something with the character if it is a TAB
   If(Character code(vtSomeText≤$vlChar≥)=Tab)
   ` ...
   End if
End for
```

8. The following example goes through the selected records for the table *[aTable]*:

```
LAST RECORD([aTable])
For($vlRecord;Records in selection([aTable]);1;-1)
   ` Do something with the record
   SEND RECORD([aTable])
   ` ...
   ` Go to the previous record
   PREVIOUS RECORD([aTable])
End for
```

## Incrementing the counter variable by more than one

If you need to, you can use an **Increment_Expression** (positive or negative) whose absolute value is greater than one.

9. The following loop addresses only the even elements of the array anArray:

```
For($vlElem;2;Size of array(anArray);2)
   ` Do something with the element #2,#4...#2n
   anArray{$vlElem}:=...
End for
```

## Getting out of a loop by changing the counter variable

In some cases, you may want to execute a loop for a specific number of iterations, but then get out of the loop when another condition becomes TRUE. To do so, you can test this condition within the loop and if it becomes TRUE, explicitly set the counter variable to a value that exceeds the end expression.

10. In the following example, a selection of the records is browsed until this is actually done or until the interprocess variable *<>vbWeStop*, intially set to FALSE, becomes TRUE. This variable is handled by an ***ON EVENT CALL*** project method that allows you to interrupt the operation:

```
<>vbWeStop:=False
ON EVENT CALL("HANDLE STOP")
  ` HANDLE STOP sets <>vbWeStop to True if Ctrl-period (Windows) or Cmd-Period (Macintosh) is pressed
$vlNbRecords:=Records in selection([aTable])
FIRST RECORD([aTable])
For($vlRecord;1;$vlNbRecords)
  ` Do something with the record
   SEND RECORD([aTable])
  ` ...
  ` Go to the next record
   If(<>vbWeStop)
      $vlRecord:=$vlNbRecords+1 ` Force the counter variable to get out of the loop
   Else
      NEXT RECORD([aTable])
   End if
End for
ON EVENT CALL("")
If(<>vbWeStop)
   ALERT("The operation has been interrupted.")
Else
   ALERT("The operation has been successfully completed.")
End if
```

## Comparing looping structures

Let's go back to the first **For...End for** example:

The following example executes 100 iterations:

```
For(vCounter;1;100)
  ` Do something
End for
```

It is interesting to see how the **While...End while** loop and **Repeat...Until** loop would perform the same action.

Here is the equivalent **While...End while** loop:

```
$i :=1 ` Initialize the counter
While($i<=100) ` Loop 100 times
  ` Do something
   $i :=$i +1 ` Need to increment the counter
End while
```

Here is the equivalent **Repeat...Until** loop:

```
$i :=1 ` Initialize the counter
Repeat
  ` Do something
   $i :=$i +1 ` Need to increment the counter
Until($i=100) ` Loop 100 times
```

**Tip**: The **For...End for** loop is usually faster than the **While...End while** and **Repeat...Until** loops, because 4D tests the condition internally for each cycle of the loop and increments the counter. Therefore, use the **For...End for** loop whenever possible.

## Optimizing the execution of the For...End for loops

You can use Real, Integer, and Long Integer variables as well as interprocess, process, and local variable counters. For lengthy repetitive loops, especially in compiled mode, use local Long Integer variables.

11. Here is an example:

```
C_LONGINT($vlCounter) ` use local Long Integer variables
For($vlCounter;1;10000)
  ` Do something
End for
```

## Nested For...End for looping structures

You can nest as many control structures as you (reasonably) need. This includes nesting **For...End for** loops. To avoid mistakes, make sure to use different counter variables for each looping structure.

Here are two examples:

12. The following example goes through all the elements of a two-dimensional array:

```
For($vlElem;1;Size of array(anArray))
  ` ...
  ` Do something with the row
  ` ...
   For($vlSubElem;1;Size of array(anArray{$vlElem}))
  ` Do something with the element
      anArray{$vlElem}{$vlSubElem}:=...
   End for
End for
```

13. The following example builds an array of pointers to all the date fields present in the database:

```
ARRAY POINTER($apDateFields;0)
$vlElem:=0
For($vlTable;1;Get last table number)
   If(Is table number valid($vlTable))
      For($vlField;1;Get last field number($vlTable))
         If(Is field number valid($vlTable;$vlField))
            $vpField:=Field($vlTable;$vlField)
            If(Type($vpField->)=Is date)
               $vlElem:=$vlElem+1
               INSERT IN ARRAY($apDateFields;$vlElem)
               $apDateFields{$vlElem}:=$vpField
            End if
         End if
      End for
   End if
End for
```

# Methods

In order to make the commands, operators, and other parts of the language work, you put them in methods. There are several kinds of methods: Object methods, Form methods, Table methods (Triggers), Project methods, and Database methods. This section describes features common to all types of methods.

A method is composed of **statements**; each statement consists of one line in the method. A statement performs an action, and may be simple or complex. Although a statement is always one line, that one line can be as long as needed (up to 32,000 characters, which is probably enough for most tasks).

For example, the following line is a statement that will add a new record to the [People] table:

```
ADD RECORD([People])
```

A method also contains **tests** and **loops** that control the flow of the execution. For a detailed discussion about the control flow programming structures, see the section **Control Flow**.

**Note**: The maximum size of a method is limited to 2 GB of text or 32 000 lines of command. Beyond these limits, a warning message appears, indicating that the extra lines will not be displayed.

## Types of Methods

There are five types of methods in 4D:

- **Object methods**: An object method is a property of an object. It is usually a short method associated with an active form object. Object methods generally “manage” the object while the form is displayed or printed. You do not call an object method—4D calls it automatically when an event involves the object to which the object method is attached.
- **Form methods**: A form method is a property of a form. You can use a form method to manage data and objects, but it is generally simpler and more efficient to use an object method for these purposes. You do not call a form method—4D calls it automatically when an event involves the form to which the form method is attached.

For more information about Object methods and Form methods, see the 4D Design Reference Manual as well as the chapter **Form Events**.

- **Table methods (Triggers)**: A Trigger is a property of a table. You do not call a Trigger. Triggers are automatically called by the 4D database engine each time that you manipulate the records of a table (Add, Delete and Modify). Triggers are methods that can prevent “illegal” operations with the records of your database. For example, in an invoicing system, you can prevent anyone from adding an invoice without specifying the customer to whom the invoice is billed. Triggers are a very powerful tool to restrict operations on a table, as well as to prevent accidental data loss or tampering. You can write very simple triggers, and then make them more and more sophisticated.

For detailed information about Triggers, see the section **Triggers**.

- **Project methods**: Unlike object methods, form methods, and triggers, which are all associated with a particular object, form, or table, project methods are available for use throughout your database. Project methods are reusable, and available for use by any other method. If you need to repeat a task, you do not have to write identical methods for each case. You can call project methods wherever you need them—from other project methods or from object or form methods. When you call a project method, it acts as if you had written the method at the location where you called it. Project methods called from other methods are often referred to as “subroutines.” A project method that returns a result can also be called a **function**.

There is one other way to use project methods—associating them with menu commands. When you associate a project method with a menu command, the method is executed when the menu command is chosen. You can think of the menu command as calling the project method.

For detailed information about Project methods, see the section **Project Methods**.

- **Database methods**: In the same way that object and form methods are called when events occur in a form, there are methods associated with the database that are called when a working session event occurs. These are the **database methods**. For example, each time you open a database, you may want to initialize some variables that will be used

during the whole working session. To do so, you use the **On Startup database method**, automatically executed by 4D when you open the database.

For more information about Database Methods, see the chapter **Database Methods**.

## An Example Project Method

All methods are fundamentally the same—they start at the first line and work their way through each statement until they reach the last line (i.e., they execute sequentially). Here is an example project method:

```
QUERY([People]) ` Display the Query editor
If(OK=1) ` The user clicked OK, not cancel
   If(Records in selection([People])=0) ` If no record was found…
      ADD RECORD([People]) ` Let the user add a new record
   End if
End if ` The end
```

Each line in the example is a **statement** or line of code. Anything that you write using the language is loosely referred to as code. Code is executed or run; this means that 4D performs the task specified by the code.

We will examine the first line in detail and then move on more quickly:

```
QUERY([People]) ` Display the Query editor
```

The first element in the line, **QUERY**, is a command. A command is part of the 4D language—it performs a task. In this case, **QUERY** displays the Query editor. This is similar to choosing Query from the Records menu in the Design environment.

The second element in the line, specified between parantheses, is an argument to the **QUERY** command. An argument (or **parameter**) is data required by a command in order to complete its task. In this case, *[People]* is the name of a table. Table names are always specified inside square brackets (*[…]*). In our example, the People table is an argument to the **QUERY** command. A command can accept several parameters.

The third element is a **comment** at the end of the line. A comment tells you (and anyone else who might read your code) what is happening in the code. It is indicated by the reverse apostrophe (`). Anything (on the line) following the comment mark will be ignored when the code is run. A comment can be put on a line by itself, or you can put comments to the right of the code, as in the example. Use comments generously throughout your code; this makes it easier for you and others to read and understand the code.

**Note**: A comment can be up to 32 000 characters long.

The next line of the method checks to see if any records were found:

```
If(Records in selection([People])=0) ` If no record was found…
```

The **If** statement is a **control-of-flow statement**—a statement that controls the step-by-step execution of your method. The **If** statement performs a test, and if the statement is true, execution continues with the subsequent lines. **Records in selection** is a function—a command that returns a value. Here, **Records in selection** returns the number of records in the current selection for the table passed as argument.

**Note**: Notice that only the first letter of the function name is capitalized. This is the naming convention for 4D functions.

You should already know what the current selection is—it is the group of records you are working on at any given time. If the number of records is equal to 0 (in other words, if no records were found), then the following line is executed:

```
ADD RECORD([People]) ` Let the user add a new record
```

The **ADD RECORD** command displays a form so that the user can add a new record. 4D formats your code automatically; notice that this line is indented to show you that it is dependent on the control-of-flow statement (If).

```
End if ` The end
```

The **End if** statement concludes the **If** statement's section of control. Whenever there is a control-of-flow statement, you need to have a corresponding statement telling the language where the control stops.

Be sure you feel comfortable with the concepts in this section. If they are all new, you may want to review them until they are clear to you.

## Where to go from here?

To learn more about:

- Object methods and Form methods, see the description of the **Form event** command as well as the 4D Design Reference manual.
- Triggers, see the section **Triggers**.
- Project methods, see the section **Project Methods**.
- Database methods, see the section **Database Methods**.

# Project Methods

Project methods are aptly named. Whereas form and object methods are bound to forms and objects, a project method is available anywhere; it is not specifically attached to any particular object of the database. A project method can have one of the following roles, depending on how it is executed and used:

- Menu method
- Subroutine and function
- Process method
- Event catching method
- Error catching method

These terms do not distinguish project methods by what they are, but by what they do.

A **menu method** is a project method called from a custom menu. It directs the flow of your application. The menu method takes control—branching where needed, presenting forms, generating reports, and generally managing your database.

The **subroutine** is a project method that can be thought of as a servant. It performs those tasks that other methods request it to perform. A function is a subroutine that returns a value to the method that called it.

A **process method** is a project method that is called when a process is started. The process lasts only as long as the process method continues to execute. For more information about processes, see the chapter **Processes**. Note that a menu method attached to a menu command whose property **Start a New Process** is selected, is also the process method for the newly started process.

An **event catching method** runs in a separate process as the process method for catching events. Usually, you let 4D do most of the event handling for you. For example, during data entry, 4D detects keystrokes and clicks, then calls the correct object and form methods so you can respond appropriately to the events from within these methods. In other circumstances, you may want to handle events directly. For example, if you run a lengthy operation (such as **For...End for** loop browsing records), you may want to be able to interrupt the operation by typing Ctrl-Period (Windows) or Cmd-Period (Macintosh). In this case, you should use an event catching method to do so. For more information, see the description of the command **ON EVENT CALL**.

An **error catching method** is an interrupt-based project method. Each time an error or an exception occurs, it executes within the process in which it was installed. For more information, see the description of the command **ON ERR CALL**.

## Menu Methods

A menu method is invoked in the Application environment when you select the custom menu command to which it is attached. You assign the method to the menu command using the Menu editor. The menu executes when the menu command is chosen. This process is one of the major aspects of customizing a database. By creating custom menus with menu methods that perform specific actions, you personalize your database. Refer to the 4D Design Reference manual for more information about the Menu editor.

Custom menu commands can cause one or more activities to take place. For example, a menu command for entering records might call a method that performs two tasks: displaying the appropriate input form, and calling the **ADD RECORD** command until the user cancels the data entry activity.

Automating sequences of activities is a very powerful capability of the programming language. Using custom menus, you can automate task sequences and thus provide more guidance to users of the database.

## Subroutines

When you create a project method, it becomes part of the language of the database in which you create it. You can then call the project method in the same way that you call 4D's built-in commands. A project method used in this way is called a subroutine.

You use subroutines to:

- Reduce repetitive coding
- Clarify your methods
- Facilitate changes to your methods

- Modularize your code

For example, let's say you have a database of customers. As you customize the database, you find that there are some tasks that you perform repeatedly, such as finding a customer and modifying his or her record. The code to do this might look like this:

```
 ` Look for a customer
QUERY BY EXAMPLE([Customers])
 ` Select the input form
FORM SET INPUT([Customers];"Data Entry")
 ` Modify the customer's record
MODIFY RECORD([Customers])
```

If you do not use subroutines, you will have to write the code each time you want to modify a customer's record. If there are ten places in your custom database where you need to do this, you will have to write the code ten times. If you use subroutines, you will only have to write it once. This is the first advantage of subroutines—to reduce the amount of code.

If the previously described code was a method called ***MODIFY CUSTOMER***, you would execute it simply by using the name of the method in another method. For example, to modify a customer's record and then print the record, you would write this method:

```
MODIFY CUSTOMER
PRINT SELECTION([Customers])
```

This capability simplifies your methods dramatically. In the example, you do not need to know how the ***MODIFY CUSTOMER*** method works, just what it does. This is the second reason for using subroutines—to clarify your methods. In this way, your methods become extensions to the 4D language.

If you need to change your method of finding customers in this example database, you will need to change only one method, not ten. This is the next reason to use subroutines—to facilitate changes to your methods.

Using subroutines, you make your code modular. This simply means dividing your code into modules (subroutines), each of which performs a logical task. Consider the following code from a checking account database:

```
FIND CLEARED CHECKS ` Find the cleared checks
RECONCILE ACCOUNT ` Reconcile the account
PRINT CHECK BOOK REPORT ` Print a checkbook report
```

Even for someone who doesn't know the database, it is clear what this code does. It is not necessary to examine each subroutine. Each subroutine might be many lines long and perform some complex operations, but here it is only important that it performs its task.

We recommend that you divide your code into logical tasks, or modules, whenever possible.

## Passing Parameters to Methods

You'll often find that you need to pass data to your methods. This is easily done with parameters.

**Parameters** (or arguments) are pieces of data that a method needs in order to perform its task. The terms parameter and argument are used interchangeably throughout this manual. Parameters are also passed to built-in 4D commands. In this example, the string "Hello" is an argument to the **ALERT** command:

```
ALERT("Hello")
```

Parameters are passed to methods in the same way. For example, if a method named ***DO SOMETHING*** accepted three parameters, a call to the method might look like this:

```
DO SOMETHING(WithThis;AndThat;ThisWay)
```

The parameters are separated by semicolons (*;*).

In the subroutine (the method that is called), the value of each parameter is automatically copied into sequentially numbered local variables: *$1*, *$2*, *$3*, and so on. The numbering of the local variables represents the order of the parameters.

The local variables/parameters are not the actual fields, variables, or expressions passed by the **calling method**; they only contain the values that have been passed.

Within the subroutine, you can use the parameters *$1*, *$2...* in the same way you would use any other local variable.

**Note**: However, in the case where you use commands that modify the value of the variable passed as parameter (for example **Find in field**), the parameters *$1*, *$2, and so on* cannot be used directly. You must first copy them into standard local variables (for example: *$myvar:=$1*).

Since they are local variables, they are available only within the subroutine and are cleared at the end of the subroutine. For this reason, a subroutine cannot change the value of the actual fields or variables passed as parameters at the calling method level. For example:

```
 ` Here is some code from the method MY METHOD
 ` ...
DO SOMETHING([People]Last Name) ` Let's say [People]Last Name is equal to "williams"
ALERT([People]Last Name)

 ` Here is the code of the method DO SOMETHING
$1:=Uppercase($1)
ALERT($1)
```

The alert box displayed by ***DO SOMETHING*** will read “WILLIAMS” and the alert box displayed by ***MY METHOD*** will read “williams”. The method locally changed the value of the parameter *$1*, but this does not affect the value of the field *[People]Last Name* passed as parameter by the method ***MY METHOD***.

There are two ways to make the method ***DO SOMETHING*** change the value of the field:

1. Rather than passing the field to the method, you pass a pointer to it, so you would write:

```
 ` Here is some code from the method MY METHOD
 ` ...
DO SOMETHING(->[People]Last Name) ` Let's say [People]Last Name is equal to "williams"
ALERT([People]Last Name)

 ` Here the code of the method DO SOMETHING
$1->:=Uppercase($1->)
ALERT($1->)
```

Here the parameter is not the field, but a pointer to it. Therefore, within the ***DO SOMETHING*** method, *$1* is no longer the value of the field but a pointer to the field. The object **referenced** by *$1* (*$1->* in the code above) is the actual field. Consequently, changing the referenced object goes beyond the scope of the subroutine, and the actual field is affected. In this example, both alert boxes will read “WILLIAMS”.

For more information about **Pointers**, see the section **Pointers**.

2. Rather than having the method ***DO SOMETHING*** “doing something,” you can rewrite the method so it returns a value. Thus you would write:

```
 ` Here is some code from the method MY METHOD
 ` ...
[People]Last Name:=DO SOMETHING([People]Last Name) ` Let's say [People]Last Name is equal to "williams"
ALERT([People]Last Name)
 ` Here the code of the method DO SOMETHING
$0:=$1
ALERT($0)
```

This second technique of returning a value by a subroutine is called “using a function.” This is described in the next paragraphs.

**Advanced note**: Parameters within the subroutine are accessible through the local variables *$1*, *$2*... In addition, parameters can be optional and can be referred to using the syntax *${...}*. For more information on parameters, see the description of the command **Count parameters**.

## Functions: Project Methods that return a value

Data can be returned from methods. A method that returns a value is called a **function**.

4D or 4D Plug-in commands that return a value are also called functions.

For example, the following line is a statement that uses the built-in function, **Length**, to return the length of a string. The statement puts the value returned by **Length** in a variable called *MyLength*. Here is the statement:

```
MyLength:=Length("How did I get here?")
```

Any subroutine can return a value. The value to be returned is put into the local variable *$0*.

For example, the following function, called ***Uppercase4***, returns a string with the first four characters of the string passed to it in uppercase:

```
$0:=Uppercase(Substring($1;1;4))+Substring($1;5)
```

The following is an example that uses the ***Uppercase4*** function:

```
NewPhrase:=Uppercase4("This is good.")
```

In this example, the variable *NewPhrase* gets "THIS is good."

The **function result**, *$0*, is a local variable within the subroutine. It can be used as such within the subroutine. For example, in the previous ***DO SOMETHING*** example, *$0* was first assigned the value of *$1*, then used as parameter to the **ALERT** command. Within the subroutine, you can use *$0* in the same way you would use any other local variable. It is 4D that returns the value of *$0* (as it is when the subroutine ends) to the called method.

## Recursive Project Methods

Project methods can call themselves. For example:

- The method A may call the method B which may call A, so A will call B again and so on.
- A method can call itself.

This is called **recursion**. The 4D language fully supports recursion.

Here is an example. Let's say you have a *[Friends and Relatives]* table composed of this extremely simplified set of fields:
- *[Friends and Relatives]Name*
- *[Friends and Relatives]ChildrensName*

For this example, we assume the values in the fields are unique (there are no two persons with the same name). Given a name, you want to build the sentence "A friend of mine, John who is the child of Paul who is the child of Jane who is the child of Robert who is the child of Eleanor, does this for a living!":

1. You can build the sentence in this way:

```
$vsName:=Request("Enter the name:";"John")
If(OK=1)
   QUERY([Friends and Relatives];[Friends and Relatives]Name=$vsName)
   If(Records in selection([Friends and Relatives])>0)
      $vtTheWholeStory:="A friend of mine, "+$vsName
      Repeat
         QUERY([Friends and Relatives];[Friends and Relatives]ChildrensName=$vsName)
         $vlQueryResult:=Records in selection([Friends and Relatives])
         If($vlQueryResult>0)
            $vtTheWholeStory:=$vtTheWholeStory+" who is the child of "+[Friends and Relatives]Name
            $vsName:=[Friends and Relatives]Name
         End if
      Until($vlQueryResult=0)
      $vtTheWholeStory:=$vtTheWholeStory+", does this for a living!"
      ALERT($vtTheWholeStory)
   End if
End if
```

2. You can also build it this way:

```
$vsName:=Request("Enter the name:";"John")
If(OK=1)
   QUERY([Friends and Relatives];[Friends and Relatives]Name=$vsName)
   If(Records in selection([Friends and Relatives])>0)
      ALERT("A friend of mine, "+Genealogy of($vsName)+", does this for a living!")
   End if
End if
```

with the recursive function ***Genealogy of*** listed here:

```
` Genealogy of project method
` Genealogy of ( String ) -> Text
` Genealogy of ( Name ) -> Part of sentence
```

```
$0:=$1
QUERY([Friends and Relatives];[Friends and Relatives]ChildrensName=$1)
If(Records in selection([Friends and Relatives])>0)
   $0:=$0+" who is the child of "+Genealogy of([Friends and Relatives]Name)
End if
```

Note the ***Genealogy of*** method which calls itself.

The first way is an **iterative algorithm**. The second way is a **recursive algorithm**.

When implementing code for cases like the previous example, it is important to note that you can always write methods using iteration or recursion. Typically, recursion provides more concise, readable, and maintainable code, but using it is not mandatory.

Some typical uses of recursion in 4D are:

- Treating records within tables that relate to each other in the same way as in the example.
- Browsing documents and folders on your disk, using the commands **FOLDER LIST** and **DOCUMENT LIST**. A folder may contain folders and documents, the subfolders can themselves contain folders and documents, and so on.

**Important**: Recursive calls should always end at some point. In the example, the method ***Genealogy of*** stops calling itself when the query returns no records. Without this condition test, the method would call itself indefinitely; eventually, 4D would return a “Stack Full” error becuase it would no longer have space to “pile up” the calls (as well as parameters and local variables used in the method).

# Debugging


- Why a Debugger?
- Syntax Error Window
- Debugger
- Watch Pane
- Call Chain Pane
- Custom Watch Pane
- Source Code Pane
- Break Points
- Break List
- Catching Commands
- Debugger Shortcuts

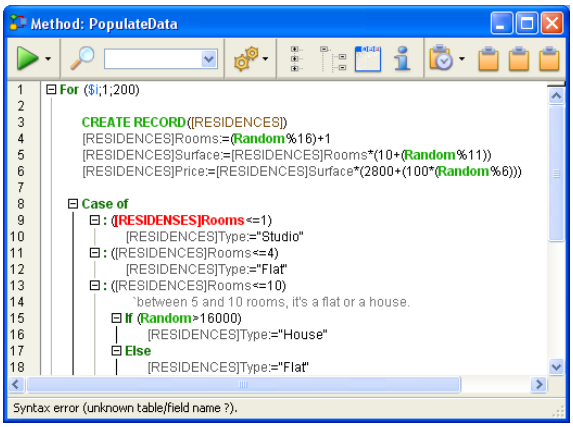
# Why a Debugger?

When developing and testing your methods, it is important that you find and fix the errors they may contain.

There are several types of errors you can make when using the language: typing errors, syntax or environmental errors, design or logic errors, and runtime errors.

## Typing Errors

Typing errors are detected by the **Method editor** and displayed in red and a message is displayed in the information area at the bottom of the method window. The following window shows a typing error:



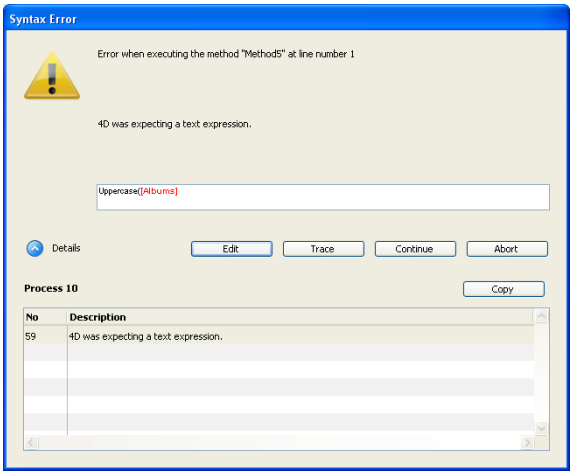

**Note**: The comments have been manually inserted for the purpose of this manual. Only the color is modified by 4D at the location of the error.

Such typing errors usually cause syntax errors (in this case, the name of the table is unknown). The information area displays a description of the error when you validate the line of code.

When this occurs, fix the typing error and type Enter (on the numeric pad) to validate the fix. For more information about the Method editor, refer to the 4D Design Reference.

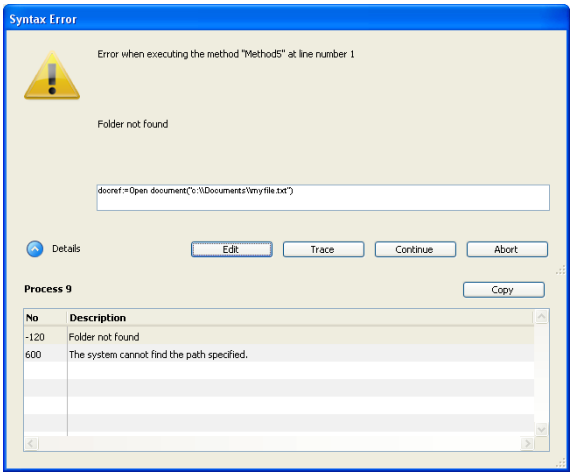
## Syntax or Environmental Errors

Some errors can be caught only when you execute the method. The **Syntax Error Window** is displayed when an error occurs. For example:




In this window, the error is that a table name is passed to the **Uppercase** command, which expects a text expression. To learn about this window and its button, see the section **Syntax Error Window**. In the above picture, the "Details" area is expanded in order to display the last error and its number.

Occasionally, there there may not be enough memory to create an array or a BLOB. When you access a document on disk, the document may not exist or may already open by another application.




These errors do not directly occur because of your code or the way you wrote it; they occur because sometimes “bad things just happen.” Most of the time, these errors are easy to treat with an error catching method installed using the command **ON ERR CALL**.

For more information about this window, refer to the **Syntax Error Window** section.

## Design or Logic Error

These are generally the most difficult type of error to find—use the **Debugger** to detect them. Note that, other than typing errors, all the previous error types are to a certain extent covered by the expression “Design or logic error.” For example:

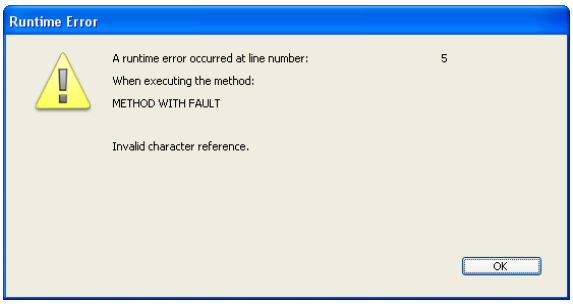
- A syntax error may occur because you try to use a variable that has not yet been initialized.
- An environmental error may occur because you try to open a document whose name is received by a subroutine which does not get the right value in the parameter. Note that in this example, the piece of code that actually “breaks” may be different than the code that is actually the origin of the problem.

Design or logic errors also include such situations as:

- A record is not properly updated because, while calling **SAVE RECORD**, you forgot to first test whether or not the record was locked.
- A method does not do exactly what you expect, because the presence of an optional parameter is not tested.

## Runtime Error

In Application mode, you can obtain errors that you never saw in interpreted mode. Here is an example:




This message indicates that you are trying to access a character whose position is beyond the length of a string. To quickly find the origin of the problem, note the name of the method and the line number, reopen the interpreted version of the structure file, and go to that method at the indicated line.

## What To Do When an Error Occurs?

Errors are common. It would be unusual to write a substantial number of lines of code (let’s say several hundred) without generating any errors. Conversely, treating and/or fixing errors is normal, too!

With its multi-tasking environment, 4D enables you to quickly edit/run methods by simply switching windows. You will discover how quickly you can fix mistakes and errors when you do not have to rerun the whole thing each time. You will also discover how quickly you can track errors if you use the **Debugger**.

A common beginner mistake in dealing with error detection is to click Abort in the Syntax Error Window, go back to the Method Editor, and try to figure out what's going by looking at the code. Do not do that! You will save plenty of time and energy by **always** using the **Debugger**.

- If an unexpected syntax error occurs, use the **Debugger**.
- If an environmental error occurs, use the **Debugger**.
- If any other type of error occurs, use the **Debugger**.

In 99% of the cases, the **Debugger** displays the information you need in order to understand why an error occurred. Once you have this information, you know how to fix the error.

**Tip**: A few hours spent in learning and experimenting with the **Debugger** can save days and weeks in the future when you have to track down errors.

Another reason to use the **Debugger** is for developing code. Sometimes you may write an algorithm that is more complex than usual. Despite all feelings of accomplishment, you are not totally sure that your coding is correct, even before trying it. Instead of running it "blind," use the **TRACE** command at the beginning of your code. Then, execute it step by step to control what happens and to check whether your suspicion was correct or not. A purist may dislike this method, but sometimes pragmatism pays off more quickly. Anyway... use the **Debugger**.

## General Conclusion

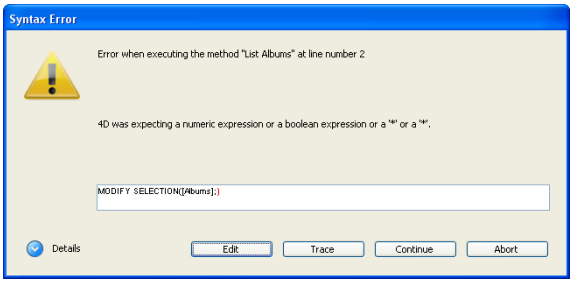
Use the **Debugger**.

# Syntax Error Window

The **Syntax Error Window** is displayed when method execution is halted. Method execution can be halted for one of the following reasons:

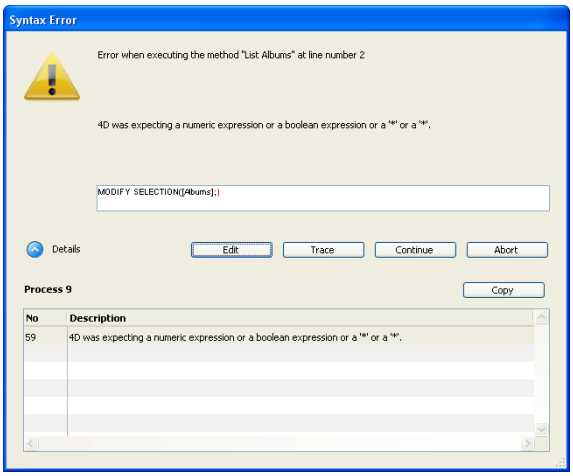
- 4D halts execution because there is an error preventing further method execution.
- The method produces a false assertion (see the **ASSERT** command).

Here is a Syntax Error window:




The upper text area of the **Syntax Error Window** displays a message describing the error. The lower text area shows the line that was executing when the error occurred; the area where the error occurred is highlighted.

The **Details** button can be used to expand the lower part of the window displaying the "stack" of errors related to the process:




There are five option buttons at the bottom of the window: **Abort**, **Trace**, **Continue**, **Edit** and (if the window is expanded) **Copy.**

- **Abort**: The method is halted, and you return to where you were before you started executing the method. If a form or object method is executing in response to an event, it is stopped and you return to the form. If the method is executing from within the Application environment, you return to this environment.
- **Trace**: You enter Trace/Debugger mode, and the **Debugger** window is displayed. If the current line has been partially executed, you may have to click the Trace button several times. Once the line finishes, you end up in the **Debugger** window.
- **Continue**: Execution continues. The line with the error may be partially executed, depending on where the error was. Continue with caution—the error may prevent the remainder of your method from executing properly. Usually, you do not want to continue. You can click **Continue** if the error is in a trivial call, such as **SET WINDOW TITLE**, which does not prevent executing and testing the rest of your code. You can thus concentrate on more important code, and fix a minor error later.
  **Note:** If you hold down the **Alt** (Windows) or **Option** (Mac OS) key, the **Continue** button changes to **Ignore**. Clicking **Ignore** means that the window will not be displayed if the same error, triggered by the same method at the same line, occurs again. This shortcut is useful in the case of an error that occurs repeatedly, for example in a loop. In this case, everything continues as if the user was clicking on the **Continue** button each time.
- **Edit**: All method execution is halted. 4D switches to the Design environment. The method in which the error occurred is opened in the Method editor, allowing you to correct the error. Use this option when you immediately recognize the

mistake and can fix it without further investigation.

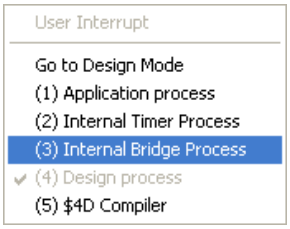
- **Copy**: This button copies the debugging information into the clipboard. This information describes the internal environment of the error (number, internal component, etc.). It is formatted as tabbed text. Once you have clicked this button, you can paste the contents of the clipboard into a text file, a spreadsheet, an e-mail, etc. for analysis purposes.

# Debugger

The term Debugger comes from the term bug. A bug in a method is a mistake that you want to eliminate. When an error has occurred, or when you need to monitor the execution of your methods, you use the debugger. A debugger helps you find bugs by allowing you to slowly step through your methods and examine method information. This process of stepping through methods is called **tracing**.

You can cause the Debugger window to display and then trace the methods in the following ways:

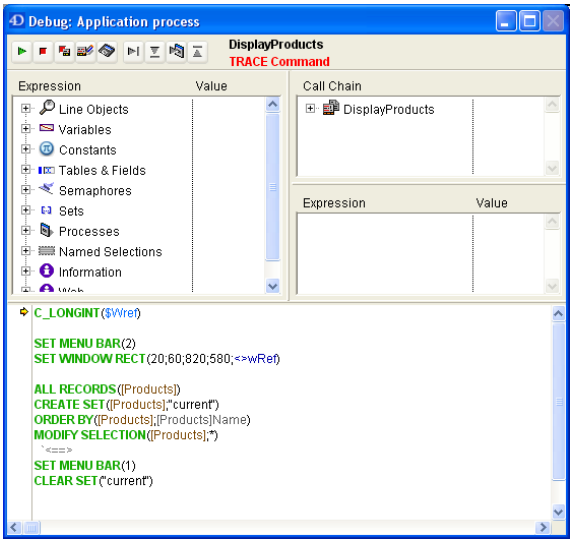
- Clicking the **Trace** button in the **Syntax Error Window**
- Using the **TRACE** command
- Clicking the **Debug** button in the Execute Method window.
- Pressing Alt+Shift+Right click (Windows) or Control+Option+Command+Click (Macintosh) while a method is executing, then selecting the process to trace in the pop-up menu:




- Clicking the **Trace** button when a process is selected in the **Process** page of the **Runtime Explorer**.
- Creating or editing a break point in the Method Editor window, or in the **Break** and **Catch** pages of the **Runtime Explorer**.

**Note**: If a password system exists for the database, only the designer and users belonging to the group that has design access privileges can trace methods.

The Debugger window is displayed here:

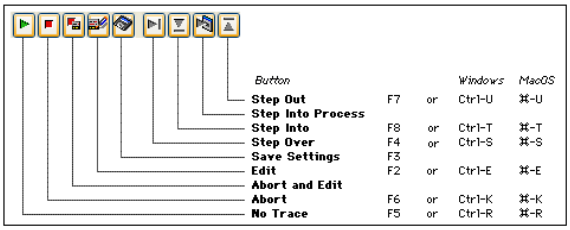



You can move the Debugger Window and/or resize any of its internal window panes as necessary. Displaying a new debug window uses the same configuration (size and position of the window, placing of the division lines and contents of the area that evaluates the expressions) as the last window displayed in the same session.

4D is a multi-tasking environment. If you run several user processes, you can trace them independently. You can have one debugger window open for each process.

## Execution Control Tool Bar Buttons

Nine buttons are located in the **Execution Control Tool Bar** at the top of the Debugger window:




### No Trace Button

Tracing is halted and normal method execution resumes.

**Note**: **Shift+F5** or **Shift+click** on the **No Trace** button resumes execution. It also disables all the subsequent **TRACE** calls for the current process.

### Abort Button

The method is halted, and you return to where you were before you started executing the method. If you were tracing a form or object method executing in response to an event, it is stopped and you return to the form. If you were tracing a method executing from within the Application environment, you return to the this environment.

### Abort and Edit Button

The method is halted as if you clicked on **Abort**. Also, 4D opens a Method Editor window for the method that was executing at the time the **Abort and Edit** button was clicked.

**Tip**: Use this button when you know which changes are required in your code and when these changes are required to pursue the testing of your methods. After you are finished with the changes, rerun the method.

### Edit Button

Clicking the Edit button does the same as Clicking **Abort and Edit** button, but does not abort the current execution. The method execution is paused at that point. 4D opens a Method Editor window for the method that was executing at the time the **Edit** button was clicked.

**Important**: You can modify this method; however, these modifications will not appear or execute in the instance of the method currently being traced in the debugger window. After the method has either aborted or completed successfully, the modifications will appear on the next execution of this method. In other words, the method must be reloaded so its modifications will be taken into account.

**Tip**: Use this button when you know which changes are required in your code and when they do not interfere with the rest of the code to be executed or traced.

**Tip**: Object Methods are reloaded for each event. If you are tracing an object method (i.e., in response to a button click), you do not need to leave the form. You can edit the object method, save the changes, then switch back to the form and retry. For tracing/changing form methods, you must exit the form and reopen it in order to reload the form method. When doing extensive debugging of a form, a trick is to put the code (that you are debugging) into a project method that you use as subroutine from within a form method. In doing so, you can stay in the form while you trace, edit, and retest your form, because the subroutine is reloaded each time it is called by the form method.

### Save Settings Button

Saves the current configuration of the debug window (size and position of the window, placing of the division lines and contents of the area that evaluates the expressions), so that it will be used by default each time the database is opened. These parameters are stored in the database's structure file.

### Step Over Button

The current method line (the one indicated by the yellow arrow—called the **program counter**) is executed, and the Debugger steps to the next line. The **Step Over** button does not step into subroutines and functions; it stays at the level of the method you are currently tracing. If you want to also trace subroutines and functions calls, use the **Step Into** button.

### Step Into Button

On execution of a line that calls another method (subroutine or function), this button causes the Debugger window to display the method being called and allows you to step through this method. The new method becomes the current (top) method in the **Call Chain Pane** of the Debugger window. On execution of a line that does not call another method, this button acts in the same manner as the **Step Over** button.

### Step Into Process Button

On execution of a line that creates a new process (i.e., calling the **New process** command), this button opens a new Debugger window that allows you to trace the process method of the newly created process. On execution of a line that does not creates a new process, this button acts in the same manner as the **Step Over** button.

### Step Out Button

If you are tracing subroutines and functions, clicking on this button allows you to execute the entire method currently being traced and to step back to the caller method. The Debugger window is brought back to the previous method in the call chain. If the current method is the last method in the call chain, the Debugger window is closed.

## Execution Control Tool Bar Information

On the right side of the execution control tool bar, the debugger provides the following information:

- The name of the method you are currently tracing (displayed in black)
- The problem caused the appearance of the Debugger window (displayed in red)

Using the example window shown above, the following information is displayed:

- The method ***DE_DebugDemo*** is the method being traced.
- The debugger window appeared because it detected a call to the **C_DATE** command and this command was one of the commands to be caught.

Here are the possible reasons for the debugger to appear and for the message (displayed in red):

- **TRACE Command**: A call to **TRACE** has been issued.
- **Break Point Reached**: A break point has been encountered.
- **User Interrupt**: You used Alt+Shift+Right click (Windows) or Control+Option+Command+Click (Macintosh), or you clicked on the **Trace** button in the **Process** page of the Design environment Runtime Explorer.
- **Caught a call to: Name of the command**: A call to a 4D command to be caught is on the point of being performed.
- **Stepping into a new process**: You used the **Step Into Process** button and this message is displayed by the Debugger window opened for the newly created process.

## The Debugger Window's Panes

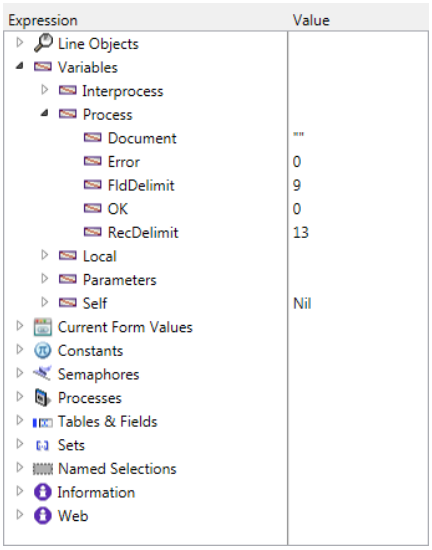
The Debugger window consists of the previously described Execution Control Tool Bar and four resizable panes:

- **Watch Pane**
- **Call Chain Pane**
- **Custom Watch Pane**
- **Source Code Pane**

The first three panes use easy-to-navigate hierarchical lists to display pertinent debugging information. The fourth one, **Source Code Pane**, displays the source code of the method being traced. Each pane has its own function to assist you in your debugging efforts. You can use the mouse to vertically and horizontally resize the debugger window and also each pane. In addition, the first three panes include a dotted separation line between the two columns they display. Using the mouse, you can move this dotted line to horizontally resize the columns, at your convenience.

# Watch Pane

The **Watch pane** is displayed in the top left corner of the Debugger window, below the Execution Control Tool Bar. Here is an example:




The **Watch Pane** displays useful general information about the system, the 4D environment, and the execution environment.

The **Expression** column displays the names of the objects or expressions. The **Value** column displays the current value of corresponding the object or expression.

Clicking on any value on the right side of the pane allows you to modify the value of the object, if this is permitted for that object.

The multi-level hierarchical lists are organized by theme at the main level. The themes are:

- Line Objects
- Variables
- Current Form Values
- Constants
- Semaphores
- Processes
- Tables & Fields
- Sets
- Named Selections
- Information
- Web

Depending on the theme, each item may have one or several sublevels. Clicking the list node next to a theme name expands or collapses the theme. If the theme is expanded, the items in that theme are visible. If the theme has several levels of information, click the list node next to each item for exploring all the information provided by the theme.

At any point, you can drag and drop themes, theme sublists (if any), and theme items to the **Custom Watch Pane**.

## Line Objects

This theme displays the values of the objects or expressions that are:

- used in the line of code to be executed (the one marked with the program counter—the yellow arrow in the **Source Code Pane**), or
- used in the previous line of code.

Since the previous line of code is the one that was just executed before, the Line Objects theme therefore shows the objects or expressions of the current line before and after that the line was executed. Let's say you execute the following method:

```
TRACE
a:=1
b:=a+1
c:=a+b
```

1. You enter the Debugger window with the **Source Code Pane** program counter set to the line *a:=1*. At this point the Line Objects theme displays:

| | |
|---|---|
| *a:* | Undefined |

The *a* variable is shown because it is used in the line to be executed (but has not yet been initialized).

2. You step one line. The program counter is now set to the line *b:=a+1*. At this point, the Line Objects theme displays:

| | |
|---|---|
| *a:* | *1* |
| *b:* | Undefined |

The *a* variable is shown because it is used in the line that was just executed and was assigned the numeric value *1*. It is also shown because it is used in the line to be executed as the expression to be assigned to the variable *b*. The *b* variable is shown because it is used in the line to be executed (but has not yet been initialized).

3. Again, you step one line. The program counter is now set to the line *c:=a+b*. At this point the Line Objects theme displays:

| | |
|---|---|
| *c:* | Undefined |
| *a:* | *1* |
| *b:* | *2* |

The *c* variable is shown because it is used in the line to be executed (but has not yet been initialized). The *a* and *b* variables are shown because there were used in the previous line and are used in the line to be executed. And so on...

The Line Objects theme is a very convenient tool—each time you execute a line, you do not need to enter an expression in the **Custom Watch Pane**, just watch the values displayed by the Line Objects theme.

## Variables

This theme is composed of the following subthemes:

- **Interprocess**: Displays the list of the interprocess variables being used at this moment. This list can be empty if you do not use interprocess variables. The values of the interprocess variables can be modified.
- **Process**: Displays the list of the process variables being used by the current process. This list is rarely empty. The values of the process variables can be modified.
- **Local**: Displays the list of the local variables being used by the method being traced (the one being shown in the *source code pane*). This list can be empty if no local variable is used or has not yet been created. The values of the local variables can be modified.
- **Parameters**: Displays the list of parameters received by the method. This list can be empty if no parameter were passed to the method being traced (the one being shown in the *source code pane*). The values of the parameters can be modified.
- **Self Pointer**: Displays a pointer to the current object if you are tracing an Object Method. This value cannot be modified

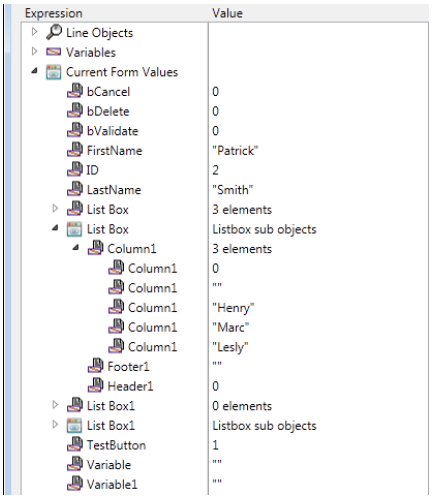
**Note**: You can modify String, Text, Numeric, Date, and Time variables; in other words, you can modify the variables whose value can be entered with the keyboard.

Arrays, like other variables, appear in the Interprocess, Process, and Locals subthemes, depending on their scope. The debugger displays each array with an additional hierarchical level; this enables you to obtain or change the values of the array elements, if any. The debugger displays the first 100 elements, including the element zero. The Value column displays the size of the array in regard to its name. After you have deployed the array, the first sub-item displays the current selected element number, then the element zero, then the other elements (up to 100). You can modify String, Text, Numeric, and Date arrays. You can modify the selected element number, the element zero, and the other elements (up to 100). You cannot modify the size of the array.

**Reminder**: At any time, you can drag and drop an item from the **Watch Pane** to the **Custom Watch Pane**, including an individual array element.

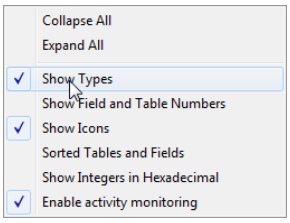
## Current Form Values

This theme contains the name of each dynamic object included in the current form, as well as the value of its associated variable:

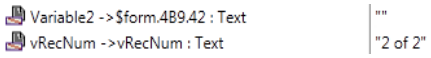



Some objects, such as list box arrays, can be presented as two distinct objects (the variable of the object itself and its data source).

This list is particularly useful when your forms use dynamic variables intensively: it is easy to identify dynamic variables through the form object names. You can display the internal name of dynamic variables by selecting **Show Types** in the context menu:




Dynamic variable names are of the "$form.4B9.42" form:




## Constants

This theme displays predefined constants provided by 4D. like the Constants page of the Explorer window. The expressions from this theme cannot be modified.

## Tables and Fields

This theme lists the tables and fields in the database; it does not list subfields. For each Table item, the Value column displays the size of the current selection for the current process as well as (if the Table item is expanded) the number of locked records. For each Field item, the Value column displays the value of the field (except picture, subtable, and BLOB) for the current record, if any. In this theme, the field values can be modified (there is no undo), but the table information cannot.

## Semaphores

This theme lists the local semaphores currently being set. For each semaphore, the Value column provides the name of the process that sets the semaphore. This list may be empty if you do not use semaphores. The expressions from this theme cannot be modified. Global semaphores are not displayed.

## Sets

This theme lists the sets defined in the current process (the one you're currently tracing); it also lists the interprocess sets. For each set, the Value column displays the number of records and the table name. This list may be empty if you do not use sets. The expressions from this theme cannot be modified.

## Processes

This theme lists the processes started since the beginning of the working session. The value column displays the time used and the current state for each process (i.e., Executing, Paused, and so on). The expressions from this theme cannot be modified.

## Named Selections

This theme lists the process named selections that are defined in the current process (the one you're currently tracing); it also lists the interprocess named selections. For each named selection, the Value column displays the number of records and the table name. This list may be empty if you do not use named selections. The expressions from this theme cannot be modified.

## Information

This theme displays general information concerning database operation, such as the current default table (if one exists), physical, virtual, free and used memory space, query destination, etc. This information allows you to examine database functioning.

## Web

This theme displays information concerning the Web server of the application (only available if the Web server is active):

- Web File To Send: name of Web file waiting to be sent (if any)
- Web Cache Usage: number of pages present in Web cache as well as its use percentage,,
- Web Server Elapsed Time: duration of Web server use in hours:minutes:seconds format
- Web Hits Count: total number of HTTP requests received since Web server launch, as well as the instantaneous number of requests per second
- Number of active Web processes: number of active Web processes, all Web processes together.

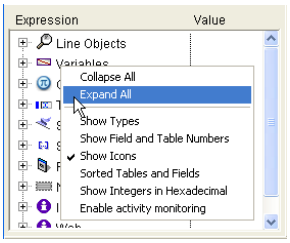
The expressions contained within this theme cannot be modified.

## Context Menu

Additional options are provided by the context menu of the Watch pane. To display this menu:

- On Windows, click anywhere in the Watch pane using the **right** mouse button.
- On Macintosh, Control-Click anywhere in the Watch pane.

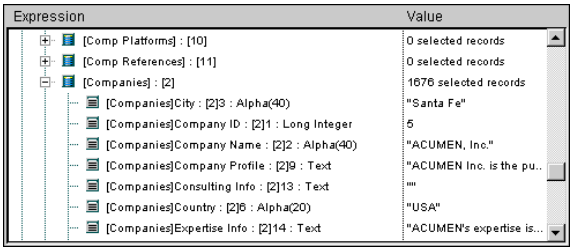
The context menu of the Watch pane is shown here:




- **Collapse All**: Collapses all levels of the Watch hierarchical list.
- **Expand All**: Expand all levels of the Watch hierarchical list.
- **Show Types**: Displays the object type for each object (when appropriate).
- **Show Field and Table Numbers**: Displays the number of each table or field of the **Fields**. If you work with table or field numbers, or with pointers using the commands such as **Table** or **Field**, this option is very useful.
- **Show Icons**: Displays an icon denoting the object type for each object. You can turn this option off in order to speed up the display, or just because you prefer to use only the **Show Types** option.
- **Sorted Tables and Fields**: Forces the table and fields to be displayed in alphabetical order, within their respective lists.

- **Show Integers in Hexadecimal**: Numbers are usually displayed in decimal notation. This option displays them in hexadecimal notation. **Note:** To enter a numeric value in hexadecimal, type 0x (zero + "x"), followed by the hexadecimal digits.
- **Enable activity monitoring**: Activates the monitoring of activity (advanced checking of internal activity of the application) and displays the information retrieved in the additional themes: **Scheduler**, **Web** and **Network**.

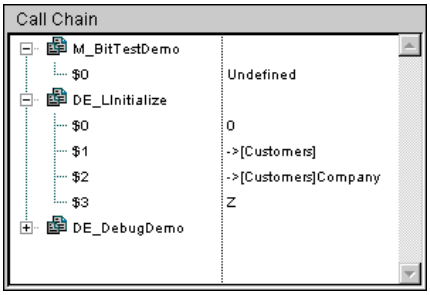
The following is a view of the **Watch Pane** with all options selected:



## Call Chain Pane

---

One method may call other methods, which may call other methods. For this reason, it is very helpful to see the chain of methods, or **Call Chain,** during the debugging process. The **Call Chain Pane**, which provides this useful function, is the top right pane of the Debugger window. This pane is displayed using a hierarchical list. Here is an example of the **Call Chain Pane**:




- Each main level item is a name of a method. The top item is the method you are currently tracing, the next main level item is the name of the caller method (the method that called the method you are currently tracing), the next one is the caller's caller method, and so on. In the example above, the method **M_BitTestDemo** is being traced; it has been called by the method **DE_LInitialize,** which has been called by **DE_DebugDemo**.

- Double-clicking the name of a method in the Call Chain pane “transports” you back to the caller method, displaying its source code in the **Source Code Pane**. In doing so, you can quickly see “how” the caller method made its call to the called method. You can examine any stage of the call chain this way.

- Clicking the node next to a Method name expands or collapses the parameter ($1, $2...) and the optional function result (*$0*) list for the method. The values appear on the right side of the pane. Clicking on any value on the right side allows you to change the value of any parameter or function result. In the figure above:

  1. **M_BitTestDemo** has not received any parameter.
  2. **M_BitTestDemo**'s *$0* is currently undefined, as the method did not assign any value to *$0* (because it has not executed this assignment yet or because the method is a subroutine and not a function).
  3. **DE_LInitialize** has received three parameters from **DE_DebugDemo**. *$1* is a pointer to the table *[Customers*], *$2* is a pointer to the field *[Customers]Company,* and *$3* is an alphanumeric parameter whose value is *"Z"*.

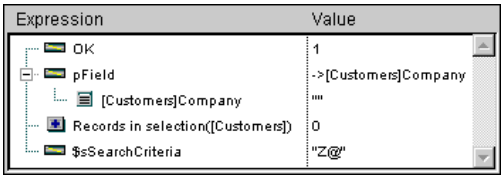
- After you have deployed the parameter list for a method, you can also drag and drop parameters and function results to the **Custom Watch Pane**.

# Custom Watch Pane

Directly below the **Call Chain Pane** is the **Custom Watch Pane**. This pane is used to evaluate expressions. Any type of expression can be evaluated, including fields, variables, pointers, calculations, built-in functions, your own functions, and anything else that returns a value.

You can evaluate any expression that can be shown in text form. This does not cover picture and BLOB fields or variables. On the other hand, the Debugger uses deployed hierarchical lists to let you display arrays and pointers. To display BLOB contents, you can use BLOB commands, such as **BLOB to text**.

In the following example, you can see several of these items: two variables, a field pointer variable and the result of a built-in function, and a calculation.




## Inserting a new expression

You can add an expression to be evaluated in the **Custom Watch Pane** in the following way:

- Drag and drop an object or expression from the **Watch Pane DISTINCT ATTRIBUTE VALUES**
- Drag and drop an object or expression from the **Call Chain Pane**
- In the **Source Code Pane** , click on an expression that can be evaluated

To create a blank expression, double-click somewhere in the empty space of the Custom Watch pane. This adds an expression ` *New expression* and then goes into editing mode so you can edit it. You can enter any 4D formula that returns a result.

After you have entered the formula, type **Enter** or **Return** (or click somewhere else in the pane) to evaluate the expression.

To change the expression, click on it to select it, then click again (or press Enter — numeric key pad) to go into editing mode.

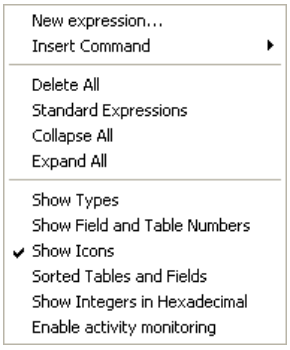
If you no longer need an expression, click on it to select it, then press **Backspace** or **Delete**.

**Warning:** Be careful when you evaluate a 4D expression modifying the value of one of the **System Variables** (for instance, the OK variable) because the execution of the rest of the method may be altered.

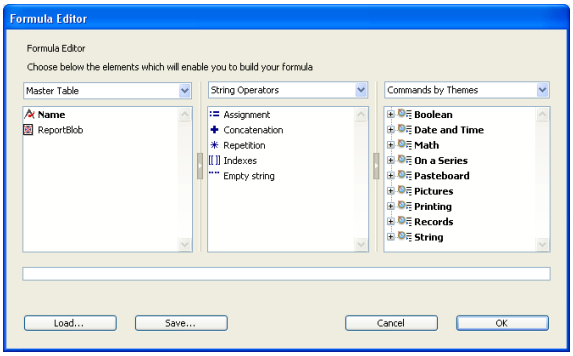
## Custom Watch Pane Context Menu

To help you enter and edit an expression, the Custom Watch Pane's context menu gives you access the 4D formula editor. In fact, the context menu also proposes additional options.

To display this menu, click anywhere in the Custom Watch pane using the **right** mouse button




- **New Expression**: This inserts a new expression and displays the 4D Formula Editor (as shown) so you can edit the new expression.




For more information about the Formula Editor, see the 4D Design Reference manual.

- **Insert Command**: This hierarchical menu item is a shortcut for inserting a command as a new expression, without using the Formula Editor.
- **Delete All**: Deletes all the expressions currently present.
- **Standard Expressions**: Recopies the list of objects in the Expression area.
- **Collapse All/Expand All**: Collapses or Expands all the expressions whose evaluation is done by the means of a hierarchical list (i.e., pointers, arrays,...)
- **Show Types**: Displays the object type for each object (when appropriate).
- **Show Field and Table Numbers**: Displays the number of each table or field of the **Fields**. If you work with table or field number or pointers using the commands such as **Table** or **Field**, this option is very useful.
- **Show Icons**: Displays an icon denoting the object type for each object. You can turn this option off in order to speed up the display, or just because you prefer to use only the **Show Types** option.
- **Sorted Tables and Fields**: Forces the table and fields to be displayed in alphabetical order, within their respective lists.
- **Show Integers in Hexadecimal**: Numbers are displayed using the decimal notation. This option displays them hexadecimal notation. **Note:** To enter a numeric value in hexadecimal, type 0x (zero + "x"), followed by the hexadecimal digits.
- **Enable activity monitoring**: Activates and displays activity monitoring information (see the **Watch Pane** section).

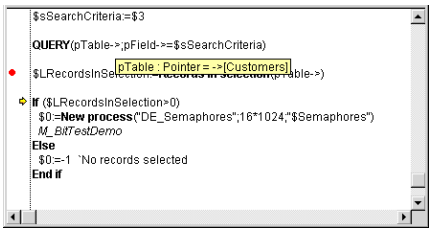
## Source Code Pane

The **Source Code Pane** shows the source code of the method being traced.

If the method is too long to fit in the text area, you can scroll to view other parts of the method.

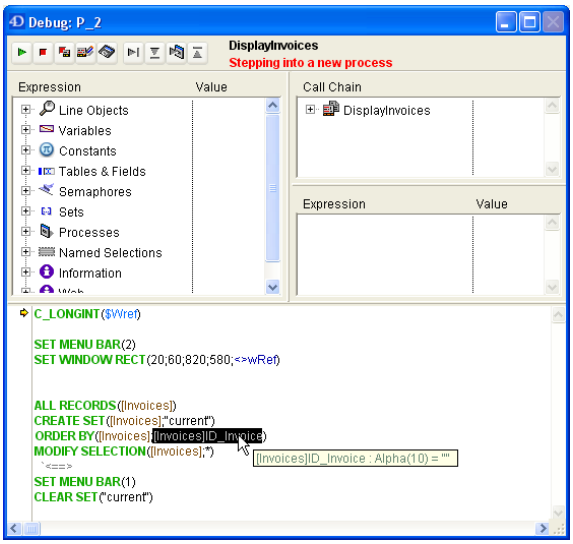
Moving the mouse pointer over any expression that can be evaluated (field, variable, pointer, array,...) will cause a **Tool Tip** to display the current value of the object or expression and its declared type.

Here is an example of the **Source Code Pane**:




A tool tip is displayed because the mouse pointer was over the variable *pTable* which, according to the tool tip, is a pointer to the table *[Customers]*.

You can also select a portion of the text in the area displaying the code being executed. In this case, when the cursor is placed above the selected text, a tip displays the selected object's value:




When you click on a variable name or field, it is automatically selected.

**Tip**: It is possible to copy any selected expression (that can be evaluated) from the **Source Code Pane** to the **Custom Watch Pane**. You can use one of the following ways:

- by simply dragging and dropping (click on the selected text, drag it and drop it in the evaluation area).
- by using the **Ctrl+D** (Windows) or **Command+D** (Mac OS) key combinations.

## Program Counter

A yellow arrow in the left margin of the Source Code pane (see the figure above) marks the next line that will be executed. This arrow is called the **program counter**. The program counter always indicates the line that is about to be executed.

For debugging purposes, you can **change** the program counter for the method being on top of the call chain (the method actually being executed). To do so, click and drag the yellow arrow vertically, to the line you want.

**WARNING**: Use this feature with caution!

Moving the program counter forward does NOT mean that the debugger is rapidly executing the lines you skip. Similarly, moving the program counter backward does NOT mean that the debugger is reversing the effect of the lines that has already been executed.

Moving the program counter simply tells the debugger to "pursue tracing or executing from here." All current settings, fields, variables, and so on are not affected by the move.

Here is an example of moving the program counter. Let's say you are debugging the following code:

```
  ` ...
If(This condition)
   DO SOMETHING
Else
   DO SOMETHING ELSE
End if
  ` ...
```

The program counter is set to the line **If (This condition)**. You step once and you see that the program counter moves to the line **DO SOMETHING ELSE**. This is unfortunate, because you wanted to execute the other alternative of the branch. In this case, and provided that the expression **This condition** does not perform operations affecting the next steps in your testing, just move the program counter back to the line **DO SOMETHING**. You can now continuing tracing the part of the code in which you are interested.

## Setting Break Points in the Debugger

In the debugging process, you may need to skip the tracing of some parts of the code. The debugger offers you several ways to execute code **up to a certain point**:

- While stepping, you can click on the **Step Over** button instead of **Step Into** button. This is useful when you do not want to enter into possible subroutines or functions called in the program counter line.
- If you mistakenly entered into a subroutine, you can execute it and directly go back to the caller method by clicking on the **Step Out** button.
- If you have a **TRACE** call placed at some point, you can click the **No Trace** button, which resumes the execution up to that call.

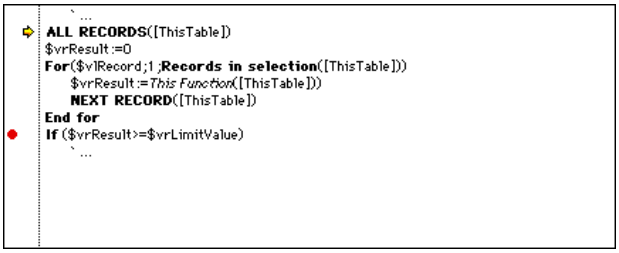
Now, let's say you are executing the following code, with the program counter set to the line *ALL RECORDS([ThisTable])*:

```
  ` ...
ALL RECORDS([ThisTable])
$vrResult:=0
For($vlRecord;1;Records in selection([ThisTable]))
   $vrResult:=This Function([ThisTable]))
   NEXT RECORD([ThisTable])
End for
If($vrResult>=$vrLimitValue)
  ` ...
```

Your goal is to evaluate the value of *$vrResult* after the For loop has been completed. Since it takes quite some execution time to reach this point in your code, you do not want to abort the current execution, then edit the method in order to insert a **TRACE** call before the line *If ($vrResult...*.

One solution is to step through the loop, however, if the table *[ThisTable]* contains several hundreds records, you are going to spend the entire day for this operation. In this type of situation, the *debugger* offers you **break points**. You can insert break points by clicking in the left margin of the Source Code pane.

For example:
You click in the left margin of the Source Code pane at the level of the line *If ($vrResult...*:




This inserts a break point for the line. The break point is indicated by a red bullet. Then click the **No Trace** button.

This resumes the normal execution **up to** the line marked with the break point. That line is not executed itself—you are back to the trace mode. In this example, the whole loop has consequently been executed normally. Then, when reaching the break point, you just need to move the mouse button over *$vrResult* to evaluate its value at the exit point of the loop.

Setting a break point beyond the program counter and clicking the **No Trace** button allows you to skip **portions** of the method being traced.

**Note**: You can also set break points directly in 4D's Method Editor. Please refer to the section **Break Points**.

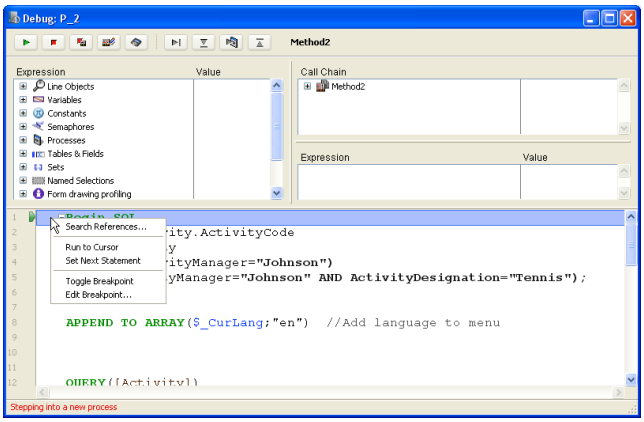
Once you add a break point, it remains associated with the method. Even if you quit the database and then reopen it later on, the break point is still there.

There are two ways to eliminate a persistent break point:

- If you are through with it, just remove it by clicking on the red bullet—the break point disappears.
- If you are not totally through with it, you may want to keep the break point. You can temporarily disable the break point by editing it. This explained in the section **Break Points**.

## Context menu of Source Code Pane

The context menu of the **Source Code Pane** provides access to several functions that are useful when executing methods in Trace mode:



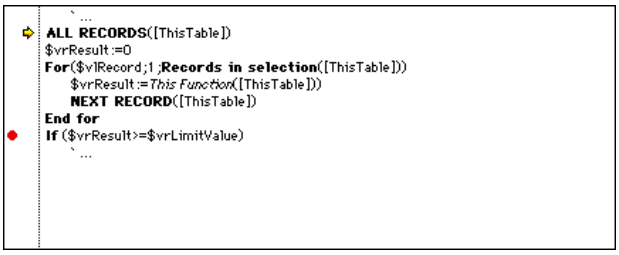

- **Goto Definition:** Goes to where the selected object is defined. This command is available for the following objects:
  - *Project methods*: displays method contents in a new window of the Method editor..
  - *Fields*: Displays field properties in the inspector of the Structure window,
  - *Tables*: Displays table properties in the inspector of the Structure window,
  - *Forms*: Displays form in the Form editor,
  - *Variables* (local, process, interprocess or $n parameter):displays the line in the current method or among the compiler methods where the variable is declared.
- **Search References** (also available in Method editor): Searches all database objects (methods and forms) in which the current element of the method is referenced. The current element is the one selected or the one where the cursor is located. This can be the name of a field, variable, command, string, and so on. Search results are displayed in a new standard results window.
- **Run to Cursor**:Executes statements found between the program counter (yellow arrow) and the selected line of the method (where the cursor is found).
- **Set Next Statement**:Moves program counter to the selected line without executing this line or any intermediate ones. The designated line is only run if the user clicks on one of the execution buttons.
- **Toggle Breakpoint** (also available in Method editor): Alternately inserts or removes the breakpoint corresponding to the selected line. This modifies the breakpoint permanently: for instance, if you remove a breakpoint in the debugger, it no longer appears in the original method.
- **Edit Breakpoint** (also available in Method editor):Displays the Breakpoint Properties dialog box. Any changes made modify the breakpoint permanently .

## Break Points

As explained in the **Source Code Pane** section, you set a break point by clicking in the left margin of the Source Code pane or of the Method Editor window, at the same level as the line of code on which you want to create the break.

**Note:** Since you can insert, modify or delete break points either in the debugger's Source Code pane or directly in the Method Editor, there is a dynamic interaction between the Method Editor and the debugger (as well as the Runtime Explorer) in regards to break points.

In the following figure, a break point has been set, in the debugger, on the line **If($vrResult>=$vrLimitValue)**:



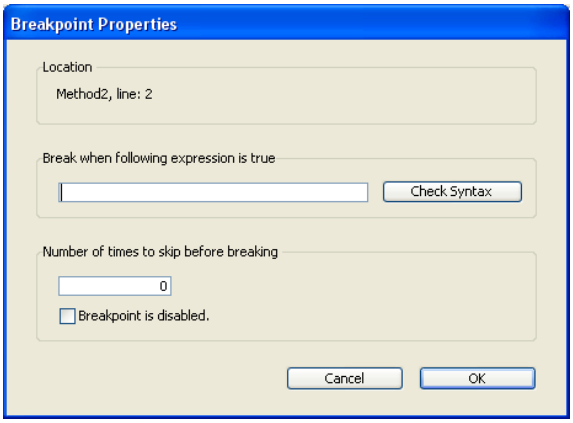

If you click again on the red bullet, the break point is deleted.

### Editing a Breakpoint

You can access the **Breakpoint Properties** window by selecting the **Edit Breakpoint** command in the context menu of the **Source Code Pane** or by pressing Alt-click (Windows) or Option-click (Macintosh) in the left margin of the window (or the Method Editor window).

- If you click on an existing break point, the window is displayed for that break point.
- If you click on a line where no break point was set, the debugger creates one and displays the window for the newly created break point.

The **Breakpoint Properties** window is shown here:




Here are the properties:

**Location**: This tells you the name of the method and the line number where the break point is set. You cannot change this information.

**Break when following expression is true**: You can create conditional break points by entering a 4D formula that returns True or False. For example, if you want to break at a line only when *Records in selection([aTable])=0*, enter this formula, and the break will occur only if there no record selected for the table *[aTable]*, when the debugger encounters the line with this break point. If you are not sure about the syntax of your formula, click the **Check Syntax** button.

**Number of times to skip before breaking**: You can set a break point to a line of code located in a loop structure (While, Repeat, or For) or located in subroutine or function called from within a loop. For example, you know that the “problem” you are tracking does not occur before at least the 200th iteration of the loop. Enter 200, and the break point will activate at the 201st iteration.

**Breakpoint is disabled**: If you currently do not need a break point, but you may need it later, you can temporarily disable the break point by editing it. A disabled break point appears as a dash (-) instead of a bullet (•) in the *source code pane* of

the debugger window, in the Method Editor and in the Break page of the Runtime Explorer.

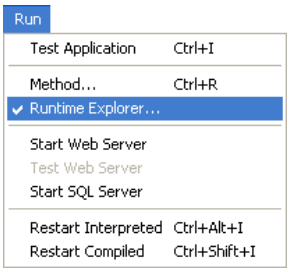
You create and edit break point from within the Debugger or the Method Editor window. You can also edit existing break points using the Break page of the Runtime Explorer. For more information, see the section **Break List**.

# Break List

The **Break List** is a page of the Runtime Explorer that enables you to manage the break points created in the Debugger Window or in the Method Editor.

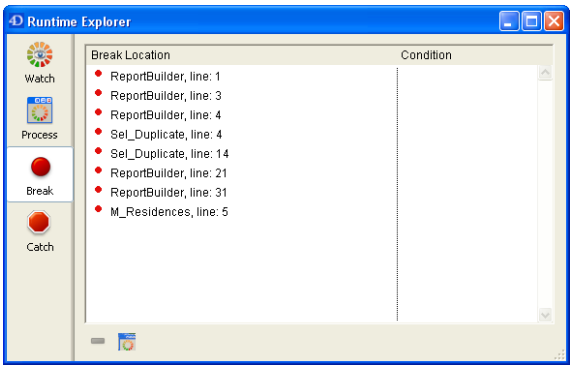
To open the Break List page:

1. Choose **Runtime Explorer** from the **Run** menu.
The Runtime Explorer can be displayed in a floating palette which always remains displayed in the front. To do this, hold down the **Shift** key while selecting **Runtime Explorer** from the **Run** menu. The Runtime Explorer is then available in all the 4D environments. For more information, please refer to the Design Reference manual.




The Runtime Explorer window appears.

2. Click on the **Break** button to display the Break List:




The Break List is composed of two columns:

- The left column displays the Enable/Disable status of the break point, followed by the name of the method and the line number where the break point has been set (using the Debugger window or the Method Editor).
- The right column displays the condition associated with the break point, if any.

Using this window, you can:

- Set a condition for a break point,
- Enable, disable or delete each break point,
- Open a Method Editor window displaying the method in which a break point is defined, by double-clicking on the break point.

However, you cannot add a new break point from this window. Break points can only be created from within the Debugger window or the Method Editor.

## Setting a Condition for a Break Point

To set a condifition for a break point, proceed as follows:

1. Click on the entry in the right column
2. Enter a 4D formula (expression or command call or project method) that returns a Boolean value.

**Note**: To remove a condition, delete its formula.

## Disabling/Enabling a Break Point

To disable or enable a break point:

1. Select the break point by clicking on it or by using the arrows to navigate through the list (if the current selected entry is not already in edit mode).
2. Choose **Enable/Disable** from the context menu.

**Shortcut**: Each entry in the list may be disabled/enabled by clicking directly on the bullet (•). The bullet changes to a dash (–) when disabled.

## Deleting a Break Point

To delete a break point:

1. Select the break point by clicking on it or by using the arrows to navigate through the list (if the current selected entry is not already in edit mode).
2. Press the **Delete** or **Backspace** key or click on the **Delete** button below the list.

**Note**: To delete all the break points, click on the **Delete All** button (second button below the list) or choose **Delete All** in the context menu.

# Catching Commands

The **Catching Commands** is a page of the Runtime Explorer that enables you to add additional breaks to your code by catching calls to 4D commands.

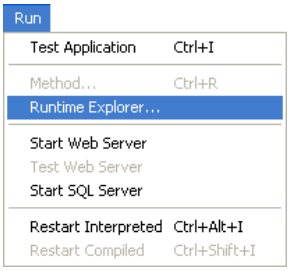
Catching a command enables you to start tracing the execution of any process as soon as a command is called by that process. Unlike a break point, which is located in a particular project method (and therefore triggers a trace exception only when it is reached), the scope of catching a command includes all the processes that execute 4D code and call that command.

Catching a command is a convenient way to trace large portions of code without setting break points at arbitrary locations. For example, if a record that should not be deleted is deleted after you have executed one or several processes, you can try to reduce the field of your investigation by catching commands such as **DELETE RECORD** and **DELETE SELECTION**. Each time these commands are called, you can check if the record in question has been deleted, and thus isolate the faulty part of the code.

With some experience, you can combine the use of Break points and command catching.

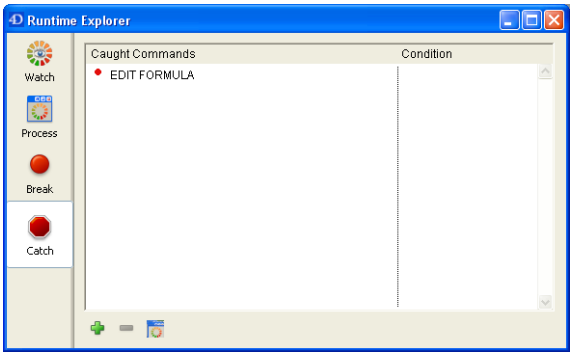
To open the Caught Commands page:

1. Choose **Runtime Explorer** from the **Run** menu.
The Runtime Explorer can be displayed in a floating palette. In this case, the floating palette always remains displayed in the front. To do this, hold down the **Shift** key while selecting **Runtime Explorer** from the **Tools** menu. For more information, please refer to the Design Reference manual.




The Runtime Explorer window appears.

2. Click on the **Catch** button to display the Caught Commands List:



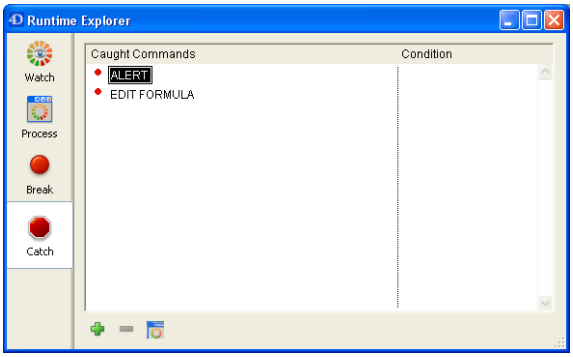

This page lists the commands to be caught during execution. It is composed of two columns:

- The left column displays the Enable/Disable status of the caught command, followed by the name of the command.
- The right column displays the condition associated with the caught command, if any.

## Adding a New Command to be Caught

To add a new command:

1. Click on the **Add New Catch** button (in the shape of a +) located below the list. A new entry is added to the list with the **ALERT** command as default.




You can then click on label **ALERT** and enter the name of the command you want to catch. Once you have finished, hit **Enter** or **Return** to validate your choice.

## Editing the Name of a Caught Command

To edit the name of a caught command:

1. Select the entry by clicking it or by using the arrow keys to navigate through the list (if the current selected entry is not already in edit mode).
2. To toggle an entry between edit mode and select mode, press **Enter** or **Return**.
3. Enter or modify the name of the command.
4. To validate your changes, press **Enter** or **Return**.

## Disabling/Enabling a Caught Command

To disable or enable a caught command:

1. Click on the bullet (•) placed in front of the command label.
   This allows you to alternately activate and/or disable the break point. The bullet's color indicates its status:
   - red = activated,
   - orange = disabled.

**Note:** Disabling a caught command has almost the same effect as deleting it. During execution, the debugger spends almost no time on the entry. The advantage of disabling an entry is that you do not have to recreate it when you need it again.

## Deleting a Caught Command

To delete a caught command:

1. Select the entry by clicking it or by using the arrow keys to navigate through the list (if the current selected entry is not already in edit mode).
2. Press the **Backspace** or **Delete** key or click on **Delete** button located beneath the list.
3. To delete all the caught commands, click on the **Delete All** button.

## Setting a Condition for Catching a Command

To set a condition for catching a command:

1. Click on the entry in the right column.
   An input cursor appears.
2. Enter a 4D formula (expression, command call or project method) that returns a Boolean value.

**Note**: To remove a condition, delete its formula.

Adding conditions allows you to stop execution when the command is invoked only if the condition is met. For example, if you associate the condition "**Records in selection**([Emp]>10)" with the break point on the **DELETE SELECTION** command, the code will not be stopped during execution of the **DELETE SELECTION** command if the current selection of the [Emp] table only contains 9 records (or less).

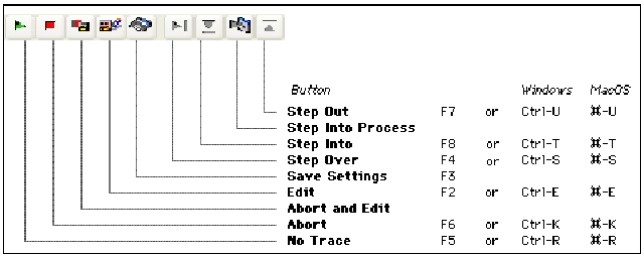
Adding conditions to caught commands slows the execution, because the condition has to be evaluated each time an exception is met. On the other hand, adding conditions accelerates the debugging process, because 4D automatically skips occurrences that do not match the conditions.

# Debugger Shortcuts

This section lists all the shortcuts available in the debugger window.

## Execution Control Tool Bar

The following figure shows the shortcuts for the nine buttons located in the top left corner of the debugger window:




Shift+F5 or Shift+click on the **No Trace** button resumes execution. Also, they disable all the next **TRACE** calls for the current process.

## Watch Pane

- Right mouse button click (Windows) or Control-Click (Macintosh) in the **Watch Pane** pulls down the Watch context menu.
- Double-click on an item of the **Watch Pane** copies the item to the **Custom Watch Pane**.

## Call Chain Pane

- Double-Click on a method name in the **Call Chain Pane** displays the method in the **Custom Watch Pane** at the line corresponding to the call in the call chain.

## Custom Watch Pane

- Right mouse button click (Windows) or Control-Click (Macintosh) in the **Custom Watch Pane** pulls down the Custom Watch context menu.
- Double-Click in the **Custom Watch Pane** creates a new watch.

## Source Code Pane

- Click in the left margin sets (persistent) or removes break points.
- **Alt-Shift-Click** (Windows) or **Option-Shift Click** (Macintosh) sets a temporary break point.
- **Alt-Click** (Windows) or **Option-Click** displays the **Edit Break** window for a new or existing break point.
- A selected expression or object can be copied to the **Custom Watch Pane** by simple drag and drop.
- **Ctrl+D** (Windows) or **Command+D** (Mac OS) key combinations copy the selected text to the **Custom Watch Pane**.

## All Panes

- **Ctrl + *** (Windows) or **Command + *** (Mac OS) forces the updating of the **Watch Pane**.
- When no item is selected in any pane, typing **Enter** steps by one line.
- When an item value is selected, use the arrows keys to navigate through the list.
- When an item is being edited, use the arrow keys to move the cursor; use **Ctrl-A/X/C/V** (Windows) or **Command-A/X/C/V** (Macintosh) as shortcuts to the **Select All/Cut/Copy/Paste** menu commands of the **Edit** menu.

# 4D Environment

- Application file
- Application type
- Application version
- BUILD APPLICATION
- Compact data file
- COMPONENT LIST
- CREATE DATA FILE
- Data file
- Get 4D file New 16.0
- Get 4D folder Updated 16.0
- Get database localization
- Get database measures
- Get database parameter
- Get last update log path
- GET SERIAL INFORMATION
- Get table fragmentation
- Is compiled mode
- Is data file locked
- NOTIFY RESOURCES FOLDER MODIFICATION
- OPEN ADMINISTRATION WINDOW
- OPEN DATA FILE
- OPEN DATABASE
- OPEN SECURITY CENTER
- OPEN SETTINGS WINDOW Updated 16.0
- PLUGIN LIST
- QUIT 4D
- RESTART 4D
- SET DATABASE LOCALIZATION
- SET DATABASE PARAMETER
- SET UPDATE FOLDER
- Structure file
- VERIFY CURRENT DATA FILE
- VERIFY DATA FILE
- Version type
- *_o_ADD DATA SEGMENT*
- *_o_DATA SEGMENT LIST*

# Application file

Application file -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String | ➲ | Long name of the 4D executable file or application |

## Description

The **Application file** command returns the long name of the 4D executable file or application you are running.

*On Windows*
If, for example, you are running 4D located at ¥PROGRAMS¥4D on the volume E, the command returns E:¥PROGRAMS¥4D¥4D.EXE.

*On Macintosh*
If, for example, you are running 4D in the Programs folder on the disk Macintosh HD, the command returns Macintosh HD:Programs:4D.app.

## Example

At startup on Windows, you need to check if a DLL Library is correctly located at the same level as the 4D executable file. In the **On Startup database method** of your application you can write:

```
If(On Windows & (Application type#4D Server))
   If(Test path name(Long name to path name(Application file)+"XRAYCAPT.DLL")#Is a document)
   ` Display a dialog box explaining that the library XRAYCAPT.DLL
   ` is missing. Therefore, the X-ray capture capability will not be available.
   End if
End if
```

**Note:** The project methods ***On Windows*** and ***Long name to path name*** are listed in the **System Documents** section.

# Application type

Application type -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Numeric value denoting the type of the application |

## Description

The **Application type** command returns a numeric value that denotes the type of 4D environment that you are running. 4D provides the following predefined constants:

| Constant | Type | Value |
|---|---|---|
| 4D Desktop | Longint | 3 |
| 4D Local mode | Longint | 0 |
| 4D Remote mode | Longint | 4 |
| 4D Server | Longint | 5 |
| 4D Volume desktop | Longint | 1 |

**Note:** *4D Desktop* incorporates certain deployment offers, such as, for example,"4D SQL Desktop".

## Example

Somewhere in your code, other than in the **On Server Startup Database Method**, you need to check if you are running 4D Server. You can write:

```
If(Application type=4D Server)
  ` Perform appropriate actions
End if
```

# Application version

Application version {( buildNum {; *} )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| buildNum | Longint | ⇦ | Build number |
| * | Operator | ⇨ | Long version number if passed, otherwise Short version number |
| Function result | String | ⇩ | Version number encoded string |

## Description

The **Application version** command returns an encoded string value that expresses the version number of the 4D environment you are running.
- If you do not pass the optional * parameter, a 4-character string is returned, formatted as follows:

| Characters | Description |
|---|---|
| 1-2 | Version number |
| 3 | "R" number |
| 4 | Revision number |

- If you pass the optional * parameter, an 8-character string is returned, formatted as follows:

| Characters | Description |
|---|---|
| 1 | "F" denotes a final version |
| | "B" denotes a beta version |
| | Other characters denote an 4D internal version |
| 2-3-4 | Internal 4D compilation number |
| 5-6 | Version number |
| 7 | "R" number |
| 8 | Revision number |

**Compatibility note (4D v14)**

Version numbering has been changed beginning with version 14 of 4D:

- the **"R" number** is the number of the "R" version of 4D, for example 3 for version R3 (contains 0 for a bug fix version),
- the **revision number** is the number of the bug fix version of 4D (contains 0 for an "R" version).

In previous versions of 4D, the number of the "R" version was the update number; it designated the revision and the revision number itself was always 0.

Examples for a short version number:

| Versions | Value returned | |
|---|---|---|
| 4D v13.1 | "1310" | *Previous numbering system* |
| 4D v14 R2 | "1420" | Release R2 |
| 4D v14 R3 | "1430" | Release R3 |
| 4D v14.1 | "1401" | First bug fix version of 4D v14 |
| 4D v14.2 | "1402" | Second bug fix version of 4D v14 |

Examples for a long version number:

| Versions | Value returned |
|---|---|
| 4D v12.5 beta | "B0011250" |
| 4D v14 R2 beta | "B0011420" |
| 4D v14 R3 final | "F0011430" |
| 4D v14.1 beta | "B0011401" |

The **Application version** command can return additional information in the optional *buildNum* parameter: the build number of the current version of the 4D application. This is an internal compilation number that can be used for versioning or when contacting the 4D Technical Services department.

**Note:** In the case of applications that are compiled and merged with 4D Volume License, the build number returned is not significant. In this context, version information is managed by the developer.

## Example 1

This example displays the 4D environment version number:

```
$vs4Dversion:=Application version
ALERT("You are using the version "+String(Num(Substring($vs4Dversion;1;2)))+"."+
$vs4Dversion≤3≥+"."+$vs4Dversion≤4≥)
```

## Example 2

This example tests to verify that you are using a final version:

```
If(Substring(Application version(*);1;1)#"F")
   ALERT("Please make sure you are using a Final Production version of 4D with this database!")
   QUIT 4D
End if
```

## Example 3

You want to use the application's short version value returned by the command to display the 4D application release name. You can write:

```
C_LONGINT($Lon_build)
C_TEXT($Txt_info;$Txt_major;$Txt_minor;$Txt_release;$Txt_version)

$Txt_version:=Application version($Lon_build)

$Txt_major:=$Txt_version[[1]]+$Txt_version[[2]] //version number, e.g. 14
$Txt_release:=$Txt_version[[3]] //Rx
$Txt_minor:=$Txt_version[[4]] //.x

$Txt_info:="4D v"+$Txt_major
If($Txt_release="0") //4D v14.x
   $Txt_info:=$Txt_info+Choose($Txt_minor#"0";"."+$Txt_minor;"")

Else //4D v14 Rx
   $Txt_info:=$Txt_info+" R"+$Txt_release
End if
```

# BUILD APPLICATION

BUILD APPLICATION {( projectName )}

| Parameter | Type | | Description |
|---|---|---|---|
| projectName | String | ⇨ | Full access path of the project to use |

## Description

The **BUILD APPLICATION** command launches the application generation process. It takes into account parameters set in the current application project or the application project set in the *projectName* parameter.

An application project is an XML file that contains all the parameters used to generate an application. Most parameters can be seen in the Build Application... dialog box (for more information, refer to the **Application builder** section of the 4D Design Reference manual).
By default, 4D creates an application project named “buildapp.xml” (default) for each database and places it in the BuildApp subfolder in the database Preferences folder.

If the database has not yet been compiled or if the compiled code is outdated, the command will first launch the compiler process. In this case, the compiler window does not appear (unless an error occurs), only a progress bar is displayed.
You can hide this progress bar using the **MESSAGES OFF** command.

If you do not pass the optional *projectName* parameter, the command displays a standard open file dialog box, so that you can designate a project file. When the dialog box has been validated, the system variable Document contains the full pathname of the open project file.

If you pass the access path and name of an XML file for a valid application project (UTF-8 encoding and “.xml” extension), the command will use the parameters defined in the file. For more information on the structure and the keys that can be used in the XML file of an application project, refer to the **4D XML Keys BuildApplication** manual.

## Example

This example builds two applications in a single method:

```
BUILD APPLICATION("c:¥¥folder¥¥projects¥¥myproject1.xml")
If(OK=1)
   BUILD APPLICATION("c:¥¥folder¥¥projects¥¥myproject2.xml")
End if
```

## System Variables or Sets

The system variable OK is set to 1 if the command has been correctly executed. Otherwise, it is set to 0. The system variable Document contains the full pathname of the open project file.

## Error Handling

If the command fails, an error is generated that you can intercept using the **ON ERR CALL** command.

# Compact data file

Compact data file ( structurePath ; dataPath {; archiveFolder {; option {; method}}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| structurePath | Text | ⇒ | Pathname of structure file |
| dataPath | Text | ⇒ | Pathname of data file to be compacted |
| archiveFolder | Text | ⇒ | Pathname of folder where original data file will be put |
| option | Longint | ⇒ | Compacting options |
| method | Text | ⇒ | Name of 4D callback method |
| Function result | Text | ⇐ | Complete pathname of folder containing original data file |

## Description

The **Compact data file** command compacts the data file designated by the *dataPath* parameter associated with the *structurePath* structure file. For more information about compacting, refer to the Design Reference manual.

To ensure the continuity of the database operation, the new compacted data file automatically replaces the original file. For security, the original file is not modified and is moved into a special folder named “Replaced files (compacting) YYYY-MM-DD HH-MM-SS” where YYYY-MM-DD HH-MM-SS represents the date and time of the backup. For example: “Replaced files (compacting) 2007-09-27 15-20-35”.

The command returns the complete pathname of the folder actually created to store the original data file. This command can only be executed from 4D (local mode) or 4D Server (stored procedure). The data file to be compacted must correspond to the structure file designated by *structurePath*. In addition, the data file must not be open when the command is executed; otherwise an error is generated.
If an error occurs during the compacting process, the original files are kept in their initial location. If an index file (.4DIndx file) is associated with the data file, it is also compacted. As with the data file, the original file is saved and the new compacted version replaces the previous one.

- In the *structurePath* parameter, pass the complete pathname of the structure file associated with the data file that you want to compact. This information is needed for the compacting procedure. The pathname must be expressed in the syntax of the operating system. You can also pass an empty string; in this case, the standard Open file dialog box appears so that you can designate the structure file to be used.
- In the *dataPath* parameter, you can pass an empty string, a file name or a complete pathname, expressed in the syntax of the operating system. If you pass an empty string, the standard Open file dialog box appears so that the user can designate the data file to be compacted. This file must correspond to the structure file defined in the *structurePath* parameter. If you only pass the name of the data file, 4D will look for it at the same level as the structure file.
- The optional *archiveFolder* parameter can be used to specify the location of the “Replaced files (compacting) DateTime” folder intended to receive the original versions of the data files as well as any index files.
  The command returns the complete pathname of the folder actually created.
  - If you omit this parameter, the original files are automatically put in a “Replaced files (compacting) DateTime” folder that is created next to the structure file.
  - If you pass an empty string, a standard Open folder dialog box will appear so that the user can specify the location of the folder to be created.
  - If you pass a pathname (expressed in the syntax of the operating system), the command will create a “Replaced files (compacting) DateTime” folder at this location.
- The optional *options* parameter is used to set various compacting options. To do so, use the following constants, found in the “**Data File Maintenance**” theme. You can pass several options by combining them:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Compact address table | Longint | 131072 | Force the address table of the records to be rewritten (slows down compacting). Note that in this case, record numbers are rewritten. If you only pass this option, 4D automatically enables the 'Update records' option. |
| Create process | Longint | 32768 | When this option is passed, compacting will be asynchronous and you will need to manage the results using the callback method (see below). 4D will not display the progress bar (it is possible to do so using the callback method). The OK system variable is set to 1 if the process has been launched correctly and 0 in all other cases. When this option is not passed, the OK variable is set to 1 if the compacting takes place correctly and 0 otherwise. |
| Do not create log file | Longint | 16384 | Generally, this command creates a log file in XML format (refer to the end of the command description). With this option, no log file will be created. |
| Timestamp log file name | Longint | 262144 | When this option is passed, the name of the log file generated will contain the date and time of its creation; as a result, it will not replace any log file already generated previously. By default, if this option is not passed, log file names are not timestamped and each new file generated replaces the previous one. |
| Update records | Longint | 65536 | Force all records to be rewritten according to current definition of the fields in the structure |

- The *method* parameter is used to set a callback method which will be called regularly during the compacting if the Create process option has been passed. Otherwise, the callback method is never called. For more information about this method, please refer to the description of the **VERIFY DATA FILE** command.
  If the callback method does not exist in the database, an error is generated and the system variable OK is set to 0.

By default, the **Compact data file** command creates a log file in XML format (if you have not passed the Do not create log file option, see the *options* parameter). This file is placed in the **Logs** folder of the current database and its name is also based on the structure file of the current database. For example, for a structure file named "myDB.4db," the log file will be named "myDB_Compact_Log.xml."
If you have passed the Timestamp log file name option, the name of the log file includes the date and time of its creation in the form "YYYY-MM-DD HH-MM-SS", which gives us, for example: "myDB_Compact_Log_2015-09-27 15-20-35.xml". This means that each new log file does not replace the previous one, but it might require subsequent manual action to remove unnecessary files.
Regardless of the option selected, as soon as a log file is generated, its path is returned in the *Document* system variable after execution of the command.

## Example

The following example (Windows) carries out the compacting of a data file:

```
$structFile:=Structure file
$dataFile:="C:¥Databases¥Invoices¥January¥Invoices.4dd"
$origFile:="C:¥Databases¥Invoices¥Archives¥January¥"
$archFolder:=Compact data file($structFile;$dataFile;$origFile)
```

## System variables and sets

If the compacting operation is carried out correctly, the OK system variable is set to 1; otherwise, it is set to 0. If a log file was generated, its complete pathname is returned in the Document system variable.

# COMPONENT LIST

COMPONENT LIST ( componentsArray )

| Parameter | Type | | Description |
|---|---|---|---|
| componentsArray | Text array | ⇦ | Names of the components |

## Description

The **COMPONENT LIST** command sizes and fills the *componentsArray* array with the names of the components loaded by the 4D application for the current host database.

When a database is opened, 4D loads the valid components found in the Components folder(s):

- the Components folder that is next to the structure file (if any),
- the Components folder that is next to the 4D application executable file.

**Reminder:** If the same component is placed in both locations, 4D will only load the one located next to the structure.

This command can be called from the host database or from a component. If the database does not use any components, the *componentsArray* array is returned empty.

The names of the components are the names of the structure files of the matrix databases (.4db, .4dc or .4dbase). This command can be used for setting up architectures and modular interfaces that offer additional functionalities according to the presence of components.

For more information about 4D components, please refer to the Design Reference manual.

# CREATE DATA FILE

CREATE DATA FILE ( accessPath )

| Parameter | Type | | Description |
|---|---|---|---|
| accessPath | String | ⇨ | Name or complete access path of the data file to create |

## Description

The **CREATE DATA FILE** command creates a new data file to disk and replaces the data file opened by the 4D application on-the-fly.

The general functioning of this command is identical to that of the **OPEN DATA FILE** command; the only difference is that the new data file set by the *accessPath* parameter is created just after the structure is re-opened.

Before launching the operation, the command verifies that the specified access path does not correspond to an existing file.

**4D Server:** Beginning with 4D v13, this command can be executed with 4D Server. In this context, it performs an internal call to **QUIT 4D** on the server (which causes a dialog box to appear on each remote machine, indicating that the server is in the process of quitting) before creating the designated file.

# Data file

Data file {( segment )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| segment | Longint | ⇒ | Obsolete, do not use |
| Function result | String | ⇄ | Long name of the data file for the database |

## Description

The **Data file** command returns the long name of the data file for the database with which you are currently working.

Starting with version 11 of 4D, data segments are no longer supported. The *segment* parameter is now ignored and must no longer be used.

*On Windows*
If, for example, you are working with the database MyCDs located at ¥DOCS¥MyCDs on the volume G, a call to **Data file** returns G:¥DOCS¥MyCDs¥MyCDs.4DD (provided that you accepted the default location and name proposed by 4D when you created the database).

*On Macintosh*
If, for example, you are working with the database located in the folder Documents:MyCDs*ƒ*: on the disk Macintosh HD, a call to **Data file** returns Macintosh HD:Documents:MyCDs*ƒ*:MyCDs.data (provided that you accepted the default location and name proposed by 4D when you created the database).

**WARNING:** If you call this command from 4D in remote mode, only the name of the data file is returned, not the long name.

# Get 4D file

Get 4D file ( file {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| file | Longint | ⇒ | File type |
| * | Operator | ⇒ | Return file path of host database |
| Function result | String | ⇐ | Pathname to 4D file |

## Description

The **Get 4D file** command returns the pathname to the 4D environment file specified by the *file* parameter. The path is returned using the system syntax.

This command allows you to get the actual pathname of specific files, whose name or location can depend on database context. It also helps you to write generic code which is independent from the 4D version or the OS.

In *file*, pass a value to specify the file for which you want to get the full pathname. You can use one of the following constants, located in the "**4D Environment**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Backup configuration file | Longint | 1 | Backup.xml file, stored in Preferences/Backup folder next to database structure file. |
| Last backup file | Longint | 2 | Last backup file, named <databaseName>[bkpNum].4BK, stored at a custom location. |
| User settings file for data | Longint | 4 | settings.4DSettings file for current data file, stored in Preferences folder next to the data file. |
| User structure settings file | Longint | 3 | settings.4DSettings file for all data files, stored in Preferences folder next to database structure file if enabled. |

When the command is called from a component, pass the optional * parameter to get the *file* path of the host database. In this case, if you omit the * parameter, an empty string is always returned.

Regarding User settings file for data and User structure settings file, a path is returned only if the **Enable User Settings in External File** security option has been checked in the "Database Settings" dialog box (see **Enabling User Settings mode**).

## Example

You want to get the path of the last backup file:

```
C_TEXT($path)
$path:=Get 4D file(Last backup file)
 // $path = "C:¥Backups¥Countries¥Countries[0025].4BK" for example
```

# Get 4D folder

Get 4D folder {( folder {; *} )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| folder | Longint | ⇒ | Folder type (if omitted = active 4D folder) |
| * | Operator | ⇒ | Return folder of host database |
| Function result | String | ⇐ | Pathname to 4D Folder |

## Description

The **Get 4D folder** command returns the pathname to the active 4D folder of the current application, or to the 4D environment folder specified by the *folder* parameter, if passed. This command allows you to get the actual pathname of the folders used by the 4D application. By using this command, you ensure that your code will work on any platform running any localized system.

In *folder*, you can pass one of the following constants, which are located in the "**4D Environment**" theme:

| Constant | Type | Value |
|---|---|---|
| 4D Client database folder | Longint | 3 |
| Active 4D Folder | Longint | 0 |
| Current resources folder | Longint | 6 |
| Data folder | Longint | 9 |
| Database folder | Longint | 4 |
| Database folder Unix syntax | Longint | 5 |
| HTML Root folder | Longint | 8 |
| Licenses folder | Longint | 1 |
| Logs folder | Longint | 7 |

You will find below a description of each folder:

**Preliminary notes about folder names:**

- {Disk} is the disk where the system is installed.
- The word User represents the name of the user that opened the session.

### Active 4D Folder

The 4D environment uses a specific folder to store the following information:

- Preferences files used by the 4D environment applications
- Shortcuts.xml file (custom keyboard shortcuts)
- Macros v2 folder (macro commands of Method editor)
- Favorites v1x folder, for example Favorites v13 (pathnames for local and remote databases that have been opened)

With the main 4D applications (4D and 4D Server), the active 4D folder is named **4D** and is created by default at the following location:

- On Windows 7 and higher: *{Disk}:¥Users¥<userName>¥AppData¥Roaming¥4D*
- On OS X: *{Disk}:Users:<userName>:Library:Application Support:4D*

Starting with 4D v13, in the case of an application merged with 4D Volume Desktop, the active 4D folder is found at the following location:

- On Windows 7 and higher: *{Disk}:¥Users¥<userName>¥AppData¥Roaming¥<databaseName>*
- On OS X: *{Disk}:Users:<userName>:Library:Application Support:<databaseName>*

### Licenses Folder

Folder containing the Licenses files of the machine.
The **Licenses** folder is placed at the following location:

- On Windows 7 and higher: *{Disk}:¥ProgramData¥4D¥Licenses*

- On OS X: *{Disk}:Library:Application Support:4D:Licenses*

**Notes:**

- In the case of an application merged with 4D Volume Desktop, the licenses folder is included in the package of the application.
- If the licenses folder cannot be created in the system because of a lack of authorization, it is created at the following locations:
  - On Windows 7 and higher: *{Disk}:¥Users¥<userName>¥AppData¥Roaming¥4D¥Licenses*
  - On OS X: *{Disk}:Users:<userName>:Library:Application Support:4D:Licenses*

### Data Folder

Path of the folder containing the current data file. The pathname is expressed using the standard syntax of the current platform.

### 4D Client Database Folder (Client machines)

4D database folder created on each 4D client machine for storing files and folders related to the database (resources, plug-ins, Resources folder, etc.).
The **4D Client Database Folder** is placed at the following location on each client machine:

- On Windows 7 and higher: *{Disk}:¥Users¥<userName>¥AppData¥Local¥4D¥<databaseName_Address>*
- On OS X: *{Disk}:Users:<userName>:Library:Caches:4D:<databaseName_Address>*

### Database Folder

Folder containing the database structure file. The pathname is expressed using the standard syntax of the current platform. With the 4D Client application, this constant is strictly equivalent to the previous 4D Client database folder constant: the command returns the pathname of the folder created locally.

### Database Folder Unix Syntax

Folder containing the database structure file. This constant designates the same folder as the previous one but the pathname returned is expressed using the Unix syntax (Posix), of the type /Users/... This syntax is mainly used when you use the **LAUNCH EXTERNAL PROCESS** command under OS X.

### Current Resources folder

Resources folder of the database. This folder contains the additional items (pictures, texts) used for the database interface. A component can have its own Resources folder. The Resources folder is located next to the database structure file.
In client/server mode, this folder can be used to organize the transfer of custom data (pictures, files, subfolders, etc.) between the server machine and the client machines. The contents of this folder are automatically updated on each client machine when it connects. All referencing mechanisms associated with the Resources folder are supported in client/server mode (.lproj folder, XLIFF, pictures, and so on). In addition, 4D provides various tools that can be used to manage and update this folder dynamically, more particularly a resources explorer.

**Note:** If the *Resources* folder does not exist for the database, executing the **Get 4D folder** command with the Current resources folder constant will create it.

### Logs Folder

The Logs folder of the database. This folder centralizes the log files of the current database. It is created at the same level as the structure file and contains the following log files:

- database conversion,
- Web server requests,
- data verification and repair,
- structure verification and repair,
- backup/restore activities journal,
- command debugging,
- 4D Server requests (generated on client machines and on the server).

**Note:** If the **Logs** folder does not exist for the database, executing the **Get 4D folder** command with the Logs folder constant will create it.

### HTML Root Folder

Current HTML root folder of the database. The pathname returned is expressed with the standard syntax of the current platform. The HTML root folder is the folder in which the 4D Web server looks for the requested Web pages and files. By default, it is named **WebFolder** and is placed next to the structure file (or its local copy in the case of 4D in remote mode).

Its location can be set on the Web/Configuration page of the Preferences or dynamically via the **WEB SET ROOT FOLDER** command.

If the **Get 4D folder** command is called from a remote 4D, the path returned is that of the remote machine, not that of 4D Server.

The optional * parameter is useful in the case of an architecture using components: it can be used to determine the database (host or component) for which you want to get the folder pathname. This parameter is only valid for Database folder, Database folder UNIX syntax and Current resources folder folders. It is ignored in all other cases.
When the command is called from a component:

- If the * parameter is passed, the command returns the pathname of the host database folder,
- If the * parameter is not passed, the command returns the pathname of the component folder.
  The database folder (Database folder and Database folder UNIX syntax) returned differs according to the type of the component architecture:
  - In the case of a .4dbase folder/package, the command returns the pathname of the .4dbase folder/package,
  - In the case of a .4db or .4dc file, the command returns the pathname of the “Components” folder,
  - In the case of an alias or shortcut, the command returns the pathname of the folder containing the original matrix database. The result differs according to the format of this database (.4dbase folder/package or .4db/.4dc file), as described above.

  When the command is called from the host database, it always returns the pathname of the host database folder, regardless of whether or not the * parameter is passed.

### Extras Folder (obsolete)

Folder with customized contents downloaded to each client machine.

**Compatibility Note:** Beginning with version 11.2 of 4D v11 SQL, it is no longer advisable to use the *Extras* folder for customized communication between the server and remote machines. It is now recommended to use the *Resources* folder for this purpose (see the description of the current *Resources* folder below). The *Extras* folder is nevertheless still supported by 4D Server so as to maintain the compatibility of existing applications.

**Note:** If the *Extras* folder does not exist for the database, executing the **Get 4D folder** command with the Extras folder constant will create it.

## Example 1

During the startup of a single-user database, you want to load (or create) your own settings in a file located in the 4D folder. To do so, in the **On Startup database method**, you can write code similar to this:

```
MAP FILE TYPES("PREF";"PRF";"Preferences file")
 ` Map PREF Mac OS file type to .PRF Windows file extension
$vsPrefDocName:=Get 4D folder+"MyPrefs" ` Build pathname to the Preferences file
 ` Check if the file exists
If(Test path name($vsPrefDocName+(".PRF"*Num(On Windows)))#Is a document)
   $vtPrefDocRef:=Create document($vsPrefDocName;"PREF") ` If not, create it
Else
   $vtPrefDocRef:=Open document($vsPrefDocName;"PREF") ` If so, open it
End if
If(OK=1)
 ` Process document contents
   CLOSE DOCUMENT($vtPrefDocRef)
Else
 ` Handle error
End if
```

## Example 2

This example illustrates the use of the Database folder UNIX syntax constant under Mac OS to list the contents of the database folder:

```
$posixpath:="¥""+Get 4D folder(Database folder Unix syntax)+"¥""
$myfolder:="ls -l "+$posixpath
$in:=""
$out:=""
$err:=""
LAUNCH EXTERNAL PROCESS($myfolder;$in;$out;$err)
```

**Note:** Under Mac OS, it is necessary to put pathnames in quotes when they contain the names of files or folders with spaces in them. The escape sequence "¥" can be used to insert the quotation mark character into the string. You can also use the statement **Char(Double quote)**.

## Example 3

You can compute the paths to the user settings files:

```
$UserSettings4Data:=Get 4D folder(Data folder)+"Preferences"+Folder separator+"settings.4DSettings"
$UserSettings:=Get 4D folder(Database folder)+"Preferences"+Folder separator+"settings.4DSettings"
```

## System variables and sets

If the *folder* parameter is invalid or if the pathname returned is empty, the OK system variable is set to 0.

# Get database localization

Get database localization {( languageType )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| languageType | Longint | ⇒ | Type of language |
| Function result | String | ⇐ | Current language of the database |

## Description

The **Get database localization** command returns the default language or the language of the database specified by the *languageType*, expressed in the standard defined by the RFC 3066. Typically, the command returns “en” for English, “es” for Spanish, etc. For more information about this standard and the values returned by this command, please refer to **MissingRef** in the *Design Reference* manual.

Several different language settings can be used simultaneously in the application. To designate the setting to be obtained, in *languageType* you can pass one of the following constants, found in the **4D Environment** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Current localization | Longint | 1 | Current language of the application: default language or language set via the **SET DATABASE LOCALIZATION** command. |
| Default localization | Longint | 0 | Language set automatically by 4D on startup according to the Resources folder and the system environment (not modifiable). |
| Internal 4D localization | Longint | 3 | Language used by 4D for sorts and text comparisons (set in the Preferences of the application). |
| User system localization | Longint | 2 | Language set by the current user of the system. |

By default, if you omit the *languageType* parameter, the command returns the default language (0).

The current language of the database can be used to determine the .lproj folder where the program will look for the localized items of the database. 4D automatically determines the current language on database startup according to the contents of the **Resources** folder and the system environment. How it works is that 4D loads the first .lproj folder of the database that corresponds to the reference language, with the following order of priority:

1. System language (under Mac OS, several languages can be set by order of preference, 4D uses this setting).
2. Language of the 4D application.
3. English
4. First language found in the **Resources** folder.

**Note:** If the database does not have an .lproj folder, 4D applies the following order of priority: 1. System language, 2. English (if the system language cannot be identified).

# Get database measures

Get database measures {( options )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| options | Object | ⇒ | Return options |
| Function result | Object | ⇐ | Object containing database measures |

## Description

The **Get database measures** command allows you to get detailed information about 4D database engine events. Returned information includes data read/write access from/to the disk or the memory cache, as well as the use of database indexes, queries and sorts.

**Get database measures** returns a single object that contains all the relevant measures. The *options* object parameter allows you to set options for the returned information.

### Overview of the returned object

The returned object contains a single property named "DB" that has the following basic structure:

```
{
    "DB": {
        "diskReadBytes": {…},
        "cacheReadBytes": {…},
        "cacheMissBytes": {…},
        "diskWriteBytes": {…},

        "diskReadCount": {…},
        "cacheReadCount": {…},
        "cacheMissCount": {…},
        "diskWriteCount": {…},

        "dataSegment1": {…},
        "indexSegment": {…},

        "tables": {…},
        "indexes": {…}

    }
}
```

This object is made up of eight properties that contain basic measures ("diskReadBytes", "cacheReadBytes", "cacheMissBytes", "diskWriteBytes", "diskReadCount", "cacheReadCount", "cacheMissCount", "diskWriteCount") and additional properties ("dataSegment1", "indexSegment", "tables", "index") that can also contain elementary properties but at a different level and with a different scope (see below).

**Note:** A property is only present inside the object if it receives contents. Properties that do not have any contents are not included in the object. For example, if the database has been opened in read-only mode and indexes have not been used, the returned object will not contain "diskWriteBytes", "diskWriteCount", "indexSegment" or "indexes".

### Elementary properties

Elementary properties can be found at different levels in the DB object. They return the same information but at different scopes. Here is a description of the elementary properties:

| Name | Information returned |
|---|---|
| diskReadBytes | Bytes read from disk |
| cacheReadBytes | Bytes read from cache |
| cacheMissBytes | Bytes missed from cache |
| diskWriteBytes | Bytes written to disk |
| diskReadCount | Read accesses from disk |
| cacheReadCount | Read accesses from cache |
| cacheMissCount | Read accesses missed from cache |
| diskWriteCount | Write accesses to disk |

The eight elementary properties all have the same object structure, for example:

```
"diskReadBytes": {     "value": 33486473620,     "history": [        // optional          {"value": 52564,"time": -1665},          {"value": 54202,"time": -1649},                ···     ] }
```

- "**value**" (number): The "value" property contains a number that represents either a quantity of bytes or a count of accesses. Basically, this value is the sum of the value(s) of the "history" object (even if the "history" object is not present).

- "**history**" (array of objects): The "history" object array is a compilation of event values grouped by second. The "history" property is present only if requested through the *options* parameter (see below). The history array will hold a maximum of 200 items. Each element of the array is itself an object that contains two properties: "value" and "time".
  - "value" (number): quantity of bytes or accesses handled during the time period designated in the associated "time" property.
  - "time" (number): number of seconds elapsed since the function has been called. In the example above ("time": -1649) means 1649 seconds ago (or more precisely between 1649 and 1650 seconds ago). During this one-second period, 54,202 bytes have been read on disk.
    The history array does not contain sequential values (-1650,-1651,-1652, etc.) The previous value is -1665, which means that nothing was read on the disk in the 15-second period between 1650 and 1665.
    **Note:** By default the array will only contain useful information.
    Since the maximum size of the array is 200, if the database is used intensively (e.g., something is read every second on the disk), the maximum length of the history will be 200 seconds. On the other hand, if almost nothing happens except, for example, once every 3 minutes, the length of the history will be 600 minutes (3*200).
    This example can be represented in the following diagram:

| 4D internal history | | Requested history: 30 | |
|---|---|---|---|
| time | value | time | value |
| -2 | 4629 | 0 | 0 |
| -4 | 7788 | -1 | 0 |
| -6 | 3718 | -2 | 4629 |
| -8 | 8814 | -3 | 0 |
| -10 | 3925 | -4 | 7788 |
| -12 | 775 | -5 | 0 |
| -14 | 6807 | -6 | 3718 |
| -16 | 3265 | -7 | 0 |
| -18 | 8086 | -8 | 8814 |
| -20 | 2539 | -9 | 0 |
| | | -10 | 3925 |
| | | -11 | 0 |
| | | -12 | 775 |
| | | -13 | 0 |
| | | -14 | 6807 |
| | | -15 | 0 |
| | | -16 | 3265 |
| | | -17 | 0 |
| | | -18 | 8086 |
| | | -19 | 0 |
| | | -20 | 2539 |
| | | -21 | -1 |
| | | -22 | -1 |
| | | -23 | -1 |
| | | -24 | -1 |
| | | -25 | -1 |
| | | -26 | -1 |
| | | -27 | -1 |
| | | -28 | -1 |
| | | -29 | -1 |
| | | -30 | -1 |

### dataSegment1 and indexSegment

The "dataSegment1" and "indexSegment" properties contain up to four elementary properties (when available):

```
"dataSegment1": {
    "diskReadBytes": {···},
    "diskWriteBytes": {···},
    "diskReadCount": {···},
    "diskWriteCount": {···}
    },
"indexSegment": {
```

```
"diskReadBytes": {…},
"diskWriteBytes": {…},
"diskReadCount": {…},
"diskWriteCount": {…}
}
```

These properties return the same information as the elementary properties, but detailed for each database file:

- "dataSegment1" represents the .4dd data file on the disk
- "indexSegment" represents the .4dx index file on the disk

For example, you can get the following object:

```
{ "DB": { "diskReadBytes": {  "value": 718260  }, "diskReadCount": {  "value": 229  }, "dataSegment1": {  "diskReadBytes": {
"value": 679092  },  "diskReadCount": {  "value": 212  }  }, "indexSegment": {  "diskReadBytes": {  "value": 39168  },
"diskReadCount": {  "value": 17  } }
```

You can figure out how it works by adding up the returned values:

*diskReadBytes.value = dataSegment1.diskReadBytes.value + indexSegment.diskReadBytes.value*
*diskWriteBytes.value = dataSegment1.diskWriteBytes.value + indexSegment.diskWriteBytes.value*
*diskReadCount.value = dataSegment1.diskReadCount.value + indexSegment.diskReadCount.value*
*diskWriteCount.value = dataSegment1.diskWriteCount.value + indexSegment.diskWriteCount.value*

### tables

The "tables" property contains as many properties as there are tables that have been accessed either in read or write mode since the opening of the database. The name of each property is the name of the table involved. For example:

```
"tables": {  "Employees": {…}  "Companies": {…}  }
```

Each table objects contains up to 12 properties:

- The first eight properties are the *elementary properties* (see above) with values related to the table involved.
- Two other properties, "records" and "blobs", also have the same eight elementary properties, but concerning only certain field types:
  - The "records" property concerns all fields of the table (strings, dates, nums, etc.) except for text, pictures and Blobs
  - The "blobs" property concerns the text, picture and Blob fields of the table.
- One or two additional properties, "fields" and "queries", may also be present depending on the queries and sorts performed on the table concerned:
  - The "fields" property contains as many "field name" attributes (which are also sub-objects) as the number of fields used for queries or sorts.
    Each field name object contains:
    - a "queryCount" object (with or without history, depending on the *options* parameter) if any query has been performed using this field
    - and/or a "sortCount" object (with or without history, depending on the *options* parameter) if any sort has been performed using this field.

    This attribute is not based on index use; all types of queries and sorts are taken into account.
    Example: Since the moment the database was launched, several queries and sorts have been carried out using the *CompID*, *Name* and *FirstName* fields. The returned object contains the following "fields" sub-object (*options* are with path and without history):

```
{  "DB": {  "tables": {  "Employees": {  "fields": {  "CompID": {
"queryCount": {  "value": 3  }  },
"Name": {  "queryCount": {  "value": 1  },
"sortCount": {  "value": 3  }  },
"FirstName": {  "sortCount": {  "value": 2  }
} (...)
```

    **Note**:The "fields" attribute is created only if a query or sort has been performed on the table; otherwise this attribute will not be present.
  - "queries" is an array of objects that provides a description of each query performed on the table. Each element of the array will contain three attributes:
    - "queryStatement" (string): query string (containing field names but not criteria values). For example: " (Companies.PK_ID != ?)"
    - "queryCount" (object):
      - "value" (number): number of times the query statement has been executed, regardless of the criteria values.
      - "history" (array of objects) (if requested in *options*): "value" and "time" standard history properties

- "duration" (object) (if the "value" is >0)
  - "value" (number): number of milliseconds
  - "history" (array of objects) (if requested in *options*): "value" and "time" standard history properties.

Example: Since the moment the database was launched, a single query has been performed on the Employees table (*options* are with path and with history):


```
{     "DB": {         "tables": {             "Employees": {                 "queries": [                     {
"queryStatement": "(Employees.Name == ?)",                         "queryCount": {                          "value": 1,
                            "history": [                                 {                                "value": 1,
                                "time": -2022                                }                            ]
},                          "duration": {                                "value": 2,                     "history": [
                                {                                        "value": 2,                             "time": -2022
                                }                                }                                       }, (...)
```


**Note:** The "queries" attribute is created when at least one query has been performed on the table.

### indexes

This is the most complex object. All tables that have been accessed using one or more of their indexes are stored as properties and, inside the properties, the names of the indexes used are also included as properties. Keyword indexes appear separately and their names are followed by "*(Keyword)*". Finally, each index name property object contains the eight elementary properties related to this index as well as up to four sub-objects depending on index use in the database since it was launched (each sub-object only exists if their corresponding operation has been performed at some point since the launch of the database).

Example: Since the moment the database was launched, several indexes of the [Employees]EmpLastName field have been solicited. In addition, 2 records were created and 16 were deleted in the [Companies] table. This table has a "name" field that is indexed. The table also has been queried and sorted using this field. The resulting object will contain:


```
"indexes": {     "Employees": {         "EmpLastName": {                 "diskReadBytes": {…},                 "cacheReadBytes": {…},
            "cacheMissBytes": {…},                 "diskWriteBytes": {…},                 "diskReadCount": {…},
"cacheReadCount": {…},                 "cacheMissCount": {…},                 "diskWriteCount": {…}             }         "EmpLastName
(Keyword)": {...},         "index3Name": {…},         "index4Name": {…},         …         }         "Companies": {             "Name":
(...)             "queryCount": {                 "value": 41             },             "sortCount": {                 "value": 3
            },             "insertKeyCount": {                 "value": 2             },             "deleteKeyCount": {
            "value": 16             }     table3Name: {…} }
```


### options parameter

The *options* parameter allows you to customize the actual information returned by the command. In *options*, you pass an object that can contain up to three properties: "withHistory", "historyLength", and "path".

| Property | Type | Description |
|---|---|---|
| "withHistory" | Boolean | "true" means the history will be returned by the function inside the returned object; "false" means the object returned by the function will not contain any history |
| "historyLength" | number | Defines the size of the returned history array in seconds(*). |
| "path" | string \| string array | Full path of specific property or array of full paths for all specific properties that you want to get. When you pass a string, only the corresponding value is returned in the "DB" object (if the path is valid). Example: "DB.tables.Employees.records.diskWriteBytes". When you pass an array of strings, all the corresponding values are returned in the "DB" object (if the paths are valid). Example: ["DB.tables.Employee.records.diskWriteBytes", "DB.tables.Employee.records.diskReadCount","DB.dataSegment1.diskReadBytes"] |

(*) As described above, the history is not stored as a sequence of seconds but only with relevant values. If nothing happens during a couple of seconds or more, nothing will be stored and a gap will appear in the internal history array. "time" can contain, for example, -2, -4, -5, -10, -15, -30 with values 200, 300, 250, 400, 500,150. If the "historyLength" property value is set to 600 (10 minutes), then the returned array will contain 0, -1, -2, -3 … -599 for time, and only the values of -2, -4, -5, -10, -15, -30 will be filled. All the other values will get 0 (zero) as a value. Also as described above, the only limit of the internal array is the size (200), not the time. This means that if there is low activity for a specific property, the oldest time can be very remote (e.g.: -3600 for one hour ago). It may also contain less than 200 values if the database was just started. In these cases, if the internal history time is more recent than the requested one OR if all the relevant values have already been set in the returned array, then the returned value will be -1.

Example: The database has just been started 20 seconds ago and the request history is 60 seconds. The returned values between 0 and -20 will be set with values or zeros, and the other ones will be set with -1. When a "-1" value is returned, this means that either the request time is too old or the value is no longer in the internal history array (i.e., the 200-item limit has been reached and older values have been removed).

### About client/server and components

This command returns information about database usage. This means that it will return a valid object with relevant values only when called:

- in 4D local mode (if called from a component, it returns information about the host database)

- on the server side in client/server mode.

If the command is called from a remote 4D, then the object will be left empty.
In this context, if you need to get information about the database on the server, the simplest way to perform this action is to create a method with the "Execute on server" option enabled.
This principle will also work for a component: if the component is used in a 4D local context, it will return information about the host database; in a 4D remote context, it will return information about the server database.

## Example 1

You want to have the history logged in the returned object:

```
C_OBJECT($param)
C_OBJECT($measures)
OB SET($param;"withHistory";True)
$measures:=Get database measures($param)
```

## Example 2

We only want to know the global number of bytes read in the cache ("cacheReadBytes"):

```
C_OBJECT($oStats)
C_OBJECT($oParams)
OB SET($oParams;"path";"DB.cacheReadBytes")
$oStats:=Get database measures($oParams)
```

The object returned contains, for example:

```
{    "DB": {        "cacheReadBytes": {            "value": 9516637        }    } }
```

## Example 3

We want to request measures for cache bytes read within the last two minutes:

```
C_OBJECT($oParams)
C_OBJECT($measures)
OB SET($oParams;"path";"DB.cacheReadBytes")
OB SET($oParams;"withHistory";True)
OB SET($oParams;"historyLength";2*60)
$measures:=Get database measures($oParams)
```

# Get database parameter

Get database parameter ( {aTable ;} selector {; stringValue} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to get the parameter, or Default table if this parameter is omitted |
| selector | Longint | ⇒ | Code of the database's parameter |
| stringValue | String | ⇐ | String value of the parameter |
| Function result | Real | ⇒ | Current value of the parameter |

## Description

The **Get database parameter** command allows you to get the current value of a 4D database parameter. When the parameter value is a character string, it is returned in the *stringValue* parameter.

The *selector* parameter designates the parameter to get. 4D offers you the following predefined constants, which are in the "**Database Parameters**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Direct2D disabled | Longint | 0 | See selector 69 (Direct2D Status) |
| Direct2D hardware | Longint | 1 | See selector 69 (Direct2D Status) |
| Direct2D software | Longint | 3 | See selector 69 (Direct2D Status) |
| Minimum Web process | Longint | 6 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 -> 32 767<br>**Description**: Minimum number of Web processes to maintain in non-contextual mode with 4D in local mode and 4D Server. By default, the value is 0 (see below). |
| Maximum Web process | Longint | 7 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 -> 32 767<br>**Description**: Maximum number of Web processes to maintain in non-contextual mode with 4D in local mode and 4D Server. By default, the value is 10.<br>In non-contextual mode, for the Web server to be reactive, 4D delays the Web processes for 5 seconds and reuses them to execute any possible future HTTP queries. In terms of performance, this is actually more advantageous than creating a new process for each query. Once a Web process is reused, it is delayed again for 5 seconds. When the maximum number of Web processes has been reached, the web process is then aborted. If no query has been attributed to a Web process within the 5 second delay, the process is aborted, except if the minimum number of Web processes has been reached (in which case the process is delayed again).<br>These parameters allow you to adjust how your Web server operates in relation to the number of requests and the memory available as well as other parameters. |
| _o_Web conversion mode | Longint | 8 | **** *Selector disabled* **** |
| _o_Database cache size | Longint | 9 | **Scope**: 4D application<br>**Kept between two sessions**: -<br>**Description**: *Constant obsolete (kept for compatibility reasons only)*. We now recommend using the **Get cache size** command. |
| 4D Local mode scheduler | Longint | 10 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Description**: see selector 12 |
| 4D Server scheduler | Longint | 11 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Description**: see selector 12 |

| Constant | Type | Value | Comment |
|---|---|---|---|
| 4D Remote mode scheduler | Longint | 12 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Possible values**: for selectors 10, 11 and 12, the *value* parameter is expressed in hexadecimal *0x00aabbcc* as follows:<br>*aa* = minimum number of ticks per call to the system (0 to 100 included).<br>*bb* = maximum number of ticks per call to the system (0 to 100 included).<br>*cc* = number of ticks between calls to the system (0 to 20 included).<br>If one of the values is out of range, 4D sets it to its maximum. You can pass one of the following preset standard values in the *value* parameter:<br>• *value* = -1: maximum priority allocated to 4D,<br>• *value* = -2: average priority allocated to 4D,<br>• *value* = -3: minimum priority allocated to 4D.<br>**Description**: This parameter allows you to dynamically set the 4D system internal calls. Depending on the Selector, the scheduler value will be set for:<br>• 4D local mode when the command is called from a 4D single-user application (*selector=10*).<br>• 4D Server when the command is called from 4D Server (*selector=11*).<br>• 4D remote mode when the command is called from a 4D connected to 4D Server (*selector=12*).<br>**Note:** The operation of selector 12 (4D Remote Mode Scheduler) differs according to whether the **SET DATABASE PARAMETER** command is executed on the server machine or on the client machine:<br>- If the command is executed on the server machine, the new value will be applied to all the client machines that connect to it subsequently.<br>- If the command is executed on the client machine, the new value is applied to the client machine immediately as well as to all the client machines that connect to the server subsequently.<br>You can use this operation to implement a dynamic and individualized management of priority for each client machine. This consists in executing the command initially on the client machine to be configured, then a second time on the server machine using the default value, which will then be used for the client machines that connect to it subsequently.<br>This operation is in effect in 4D starting with versions 6.8.6, 2003.3 and 2004.<br>**Warning**: Configuring these selectors inappropriately can cause serious degradation of application performance. It is recommended to only modify the default values with full knowledge of the facts. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| 4D Server timeout | Longint | 13 | **Scope**: 4D application if *value* positive<br>**Kept between two sessions**: Yes if *value* positive<br>**Possible values**: 0 -> 32 767<br>**Description**: Value of the 4D Server timeout. The default 4D Server timeout value is defined on the "Client-Server/Network options" page of the Database settings dialog box on the server side.<br>The server timeout sets the maximum period "authorized" to wait for a client response, for example when it is executing a blocking operation. After this period, 4D Server disconnects the client. The 4D Server Timeout selector allows you to set, in the corresponding *value* parameter, a new timeout expressed in minutes. This feature is particularly useful to increase the timeout before executing a blocking and time-consuming operation on the client, such as printing a large number of pages, which can cause an unexpected timeout.<br>You also have two options:<br>• If you pass a **positive** value in the *value* parameter, you set a global and permanent timeout: the new value is applied to all process and is stored in the preferences of the 4D application (equivalent to change in the Preferences dialog box).<br>• If you pass a **negative** value in the *value* parameter, you set a local and temporary timeout: The new value is applied to the calling process only (the other processes keep the default values) and is reset to default as soon as the server receives any signal of activity from the client — for example, when the operation is finished. This option is useful for managing long operations initiated by 4D plug-ins.<br>To set the "No timeout" option, pass 0 in *value*. See example 1. |
| 4D Remote mode timeout | Longint | 14 | **Scope** (legacy network layer only): 4D application if *value* positive<br>**Kept between two sessions**: Yes if *value* positive<br>**Description**: To be used in very specific cases. Value of the timeout granted by the remote 4D machine to the 4D Server machine. The default timeout value used by 4D in remote mode is set on the "Client-Server/Network options" page of the Database settings dialog box on the remote machine.<br>The 4D Remote mode timeout selector is only taken into account if you are using the legacy network. It is ignored when the *ServerNet* layer is activated: this setting is entirely managed by the 4D Server timeout (13) selector. |
| Port ID | Longint | 15 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: No<br>**Description**: TCP port ID used by the 4D Web server with 4D in local mode and 4D Server. The default value, which can be set on the "Web/Configuration" page of the Preferences dialog box, is 80. You can use the constants of the **TCP Port Numbers** theme for the *value* parameter.<br>The Port ID selector is useful for 4D Web Servers compiled and merged with 4D Desktop (in which there is no access to the Design mode). For more information about the TCP port ID, refer to the **Web Server Settings** section. |
| IP Address to listen | Longint | 16 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Character set | Longint | 17 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Max concurrent Web processes | Longint | 18 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Client minimum Web process | Longint | 19 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 6<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client maximum Web process | Longint | 20 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 7<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client Max Web requests size | Longint | 21 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 27<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client port ID | Longint | 22 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 15<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client IP address to listen | Longint | 23 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 16<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client character set | Longint | 24 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 17<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client max concurrent Web proc | Longint | 25 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 18<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Maximum Web requests size | Longint | 27 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| 4D Server log recording | Longint | 28 | **Scope**: 4D Server, 4D remote<br>**Kept between two sessions**: No<br>**Possible values**: 0 or from 1 to X (0 = do not record, 1 to X = sequential number, added to the file name).<br>**Description**: Starts or stops the recording of standard requests received by 4D Server (excluding Web requests). By default, the value is 0 (requests not recorded).<br>4D Server lets you record each request received by the server machine in a log file. When this mechanism is enabled, two files are created in the Logs folder of the database, next to the database structure file.They are named 4DRequestsLog_X.txt and 4DRequestsLog_ProcessInfo_X.txt, where X is the sequential number of the log. Once the file 4DRequestsLog has reached a size of 10 MB, it is closed and a new one is generated, with an incremented sequential number. If a file of the same name already exists, it is replaced directly. You can set the starting number of the sequence using the *value* parameter.<br>These text files store various information concerning each request in a simple tabbed format: time, process number, size of request, processing duration, etc. This information is particularly useful during the development phase of the application or for statistical purposes. It can be imported, for example, into a spreadsheet software in order to be processed. |
| _o_Web Log recording | Longint | 29 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Client Web log recording | Longint | 30 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 = Do not record (default), 1 = Record in CLF format, 2 = Record in DLF format, 3 = Record in ELF format, 4 = Record in WLF format.<br>**Description**: Starts or stops the recording of Web requests received by the Web servers of all the client machines. By default, the value is 0 (requests not recorded).<br>The operation of this selector is identical to that of selector 29; however, it applies to all the 4D remote machines used as Web servers. The "logweb.txt" file is, in this case, automatically placed in the Logs subfolder of the remote 4D database folder (cache folder). If you only want to set values for certain client machines, use the Preferences dialog box of 4D in remote mode. |
| Table sequence number | Longint | 31 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Possible values**: Any longint value.<br>**Description**: This selector is used to modify or get the current unique number for records of the table passed as parameter. "Current number" means "last number used": if you modify this value using **SET DATABASE PARAMETER**, the next record will be created with a number that consists of the value passed + 1. This new number is the one returned by the **Sequence number** command as well in any field of the table to which the "Autoincrement" property has been assigned in the Structure editor or via SQL.<br>By default, this unique number is set by 4D and corresponds to the order of record creation. For additional information, refer to the documentation of the **Sequence number** command. |
| _o_Real display precision | Longint | 32 | **** *Selector disabled* **** |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Debug log recording | Longint | 34 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Description**: Starts or stops the sequential recording of events occurring at the 4D programming level in the *4DDebugLog* file, which is automatically placed in the Logs subfolder of the database, next to the structure file. A new, more compact, tabbed text format is used in the event log file "4DDebugLog[_n].txt" starting with 4D v14 (where _n is the segment number of the file).<br>**Possible values**: Longint containing a bit field: value = bit1(1)+bit2(2)+bit3(4)+bit4(8)+⋯).<br>- Bit 1 (value 1) requests to enable the file (note that any other non-null value also enables it as well)<br>- Bit 2 (value 2) requests call parameters to methods and commands.<br>- Bit 3 (value 4) enables new tabbed format.<br>- Bit 4 (value 8) disables immediate writing of each operation on disk (enabled by default). Immediate writing is slower but more effective, for example for investigating causes of a crash. If you disable this mode, the file contents are more compact and are generated more quickly.<br>- Bit 5 (value 16) disables recording of plug-in calls (enabled by default).<br>In the (former) non-tabbed format, execution times are expressed in milliseconds and the "< ms" value is displayed when an operation lasts less than one millisecond.<br>In the new tabbed format, execution times are expressed in microseconds.<br>Examples:<br>SET DATABASE PARAMETER (34;1) // enables mode v13 file without parameters, with runtimes<br>SET DATABASE PARAMETER (34;2) // enables mode v13 file with parameters and runtimes<br>SET DATABASE PARAMETER (34;2+4) // enables file with v14 format, with parameters and runtimes<br>SET DATABASE PARAMETER (34;0) // disables file<br>To avoid having a file record too much information, you can restrict the 4D commands that are examined by using selector 80, Log Command list.<br>This option can be enabled for any type of 4D application (4D all modes, 4D Server, 4D Volume Desktop), in interpreted or compiled mode.<br>**Note:** This option is provided solely for the purpose of debugging and must not be put into production since it may lead to deterioration of the application performance and saturation of the hard disk. For more information about this format and on the use of the 4DDebugLog[_n].txt file, please contact the Technical Support of 4D Inc. |
| Client Server port ID | Longint | 35 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 to 65535<br>**Description**: TCP port number where the 4D Server publishes the database (bound for 4D remote machines). By default, the value is 19813.<br>Customizing this value means that several 4D client-server applications can be used on the same machine with the TCP protocol; in this case, you must indicate a different port number for each application.<br>The value is stored in the database structure file. It can be set with 4D in local mode but is only taken into account in client-server configuration.<br>When you modify this value, it is necessary to restart the server machine in order for the new value to be taken into account. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Invert objects | Longint | 37 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0, 1 or 2 (0 = mode disabled, 1 = automatic mode, 2 = mode enabled).<br>**Description**: Configuration of the "object inversion" mode which is used to invert forms, objects, menu bars, and so on, in Application mode when the database is displayed under Windows in a right-to-left language. This mode can also be configured on the Interface/Right-to-left languages page of the Database Settings.<br>• Value 0 indicates that the mode is never enabled, regardless of the system configuration (corresponds to the Never value in the Database Settings).<br>• Value 1 indicates that the mode is enabled or disabled depending on the system configuration (corresponds to the Automatic value in the Database Settings).<br>• Value 2 indicates that the mode is enabled, regardless of the system configuration (corresponds to the Always value in the Database Settings).<br>For more information, refer to the *Design Reference* manual of 4D. |
| HTTPS Port ID | Longint | 39 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Client HTTPS port ID | Longint | 40 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 to 65535<br>**Description**: TCP port number used by the Web servers of the client machines for secure connections via SSL (HTTPS protocol). By default, the value is 443 (standard value).<br>This selector can be used to modify by programming the TCP port used by the Web servers of the client machines for secure connections via SSL (HTTPS protocol). By default, the value is 443 (standard value).<br>This selector operates exactly the same way as selector 39; however, it applies to all the 4D remote machines used as Web servers. If you only want to modify the value of certain specific client machines, use the Preferences dialog box of the remote 4D. |
| Unicode mode | Longint | 41 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (compatibility mode) or 1 (Unicode mode)<br>**Description**: Current database operating mode, with regards to the character set. 4D supports the Unicode character set but can function in “compatibility” mode (based on the Mac ASCII character set). By default, converted databases are executed in compatibility mode (0) and databases created with version 11 or higher are executed in Unicode mode. The execution mode can be controlled via an option in the Preferences and can also be read or (for testing purposes) modified via this selector. Modifying this option requires the database to be restarted in order for it to be taken into account. Note that within a component it is not possible to modify this value, but only to read it. |
| SQL Autocommit | Longint | 43 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (deactivation) or 1 (activation)<br>**Description**: Activation or deactivation of the SQL auto-commit mode. By default, the value is 0 (deactivated mode)<br>The auto-commit mode is used to strengthen the referential integrity of the database. When this mode is active, all **SELECT**, **INSERT**, **UPDATE** and **DELETE** (SIUD) queries are automatically included in ad hoc transactions when they are not already executed within a transaction. This mode can also be set in the Preferences of the database. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| SQL Engine case sensitivity | Longint | 44 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (case not taken into account) or 1 (case-sensitive)<br>**Description**: Activation or deactivation of case-sensitivity for string comparisons carried out by the SQL engine.<br>By default, the value is 1 (case-sensitive): the SQL engine differentiates between upper and lower case and between accented characters when comparing strings (sorts and queries). For example “ABC”= “ABC” but “ABC” # “Abc” and "abc" # "âbc." In certain cases, for example so as to align the functioning of the SQL engine with that of the 4D engine, you may wish for string comparisons to not be case-sensitive (“ABC”=“Abc"="âbc").<br>This option can also be set on the **SQL page** of the Database settings. |
| Client log recording | Longint | 45 | **Scope**: Remote 4D machine<br>**Kept between two sessions**: No<br>**Possible values**: 0 or from 1 to X (0 = do not record, 1 to X = sequential number, attached to file name).<br>**Description**: Starts or stops recording of standard requests carried out by the 4D client machine that executed the command (excluding Web requests). By default, the value is 0 (no recording of requests).<br>4D lets you record the log of requests carried out by the client machine. When this mechanism is activated, two files are created on the client machine, in the Logs subfolder of the local folder of the database. They are named 4DRequestsLog_X.txt and 4DRequestsLog_ProcessInfo_X.txt, where X is the sequential number of the log. Once the file 4DRequestsLog has reached a size of 10 MB, it is closed and a new one is generated, with an incremented sequential number. If a file with the same name already exists, it is directly replaced. You can set the starting number for the sequence using the value parameter.<br>These text files store various information concerning each request in a simple tabbed format: time, process number, size of request, processing duration, etc. This information is particularly useful during the development phase of the application or for statistical purposes |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Query by formula on server | Longint | 46 | **Scope**: Current table and process<br>**Kept between two sessions**: No<br>**Possible values**: 0 (use database configuration), 1 (execute on client) or 2 (execute on server)<br>**Description**: Execution location of **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** commands for the *table* passed in the parameter.<br>When using a database in client-server mode, the query "by formula" commands can be executed either on the server or on the client machine:<br>• In databases created with 4D v11 SQL, these commands are executed on the server.<br>• In converted databases, these commands are executed on the client machine, as in previous versions of 4D.<br>• In converted databases, a specific preference (Application/Compatibility page) can be used to globally modify the execution location of these commands.<br>This difference in execution location influences not only application performance (execution on the server is usually faster) but also programming. In fact, the value of the components of the formula (in particular variables called via a method) differ according to the execution context. You can use this selector to punctually adapt the operation of your application.<br>If you pass 0 in the *value* l'parameter, the execution location of query "by formula" commands will depend on the database configuration: in databases created with 4D v11 SQL, these commands will be executed on the server. In converted databases, they will be executed on the client machine or the server according to the database preferences. Pass 1 or 2 in *value* to "force" the execution of these commands, respectively, on the client or on the server machine.<br>Refer to example 4.<br>**Note:** If you want to be able to enable "SQL type" joins (see the QUERY BY FORMULA Joins selector), you must always execute formulas on the server so that they have access to the records. Be careful, in this context, the formula must not contain any calls to a method, otherwise it will automatically be switched to the remote machine. |
| Order by formula on server | Longint | 47 | **Scope**: Current table and process<br>**Kept between two sessions**: No<br>**Possible values**: 0 (use database configuration), 1 (execute on client) or 2 (execute on server)<br>**Description** : Execution location of **ORDER BY FORMULA** command for the table passed in the parameter.<br>When using a database in client-server mode, this command can be executed either on the server or on the client machine. This selector can be used to specify the execution location of this command (server or client). This mode can also be set in the database preferences. For more information, please refer to the description of selector 46, Query By Formula On Server.<br>**Note:** If you want to be able to enable "SQL type" joins (see the QUERY BY FORMULA Joins selector), you must always execute formulas on the server so that they have access to the records. Be careful, in this context, the formula must not contain any calls to a method, otherwise it will automatically be switched to the remote machine. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Auto synchro resources folder | Longint | 48 | **Scope**: 4D remote machine<br>**Kept between two sessions**: No<br>**Possible values**: 0 (no synchronization), 1 (auto synchronization) or 2 (ask).<br>**Description**: Dynamic synchronization mode for *Resources* folder of 4D client machine that executed the command with that of the server.<br>When the contents of the *Resources* folder on the server has been modified or a user has requested synchronization (for example via the resources explorer or following the execution of the **SET DATABASE LOCALIZATION** command), the server notifies the connected client machines.<br>Three synchronization modes are then possible on the client side. The Auto Synchro Resources Folder selector is used to specify the mode to be used by the client machine for the current session:<br>• 0 (default value): no dynamic synchronization (synchronization request is ignored)<br>• 1: automatic dynamic synchronization<br>• 2: display of a dialog box on the client machines, with the possibility of allowing or refusing synchronization.<br>The synchronization mode can also be set globally in the application Preferences. |
| Query by formula joins | Longint | 49 | **Scope**: Current process<br>**Kept between two sessions**: No<br>**Possible values**: 0 (use database configuration), 1 (always use automatic relations) or 2 (use SQL joins if possible).<br>**Description**: Operating mode of the **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** commands relating to the use of "SQL joins."<br>In databases created starting with version 11.2 of 4D v11 SQL, these commands carry out joins based on the SQL joins model. This mechanism can be used to modify the selection of a table according to a query carried out on another table without these tables being connected by an automatic relation (necessary condition in previous versions of 4D).<br>The QUERY BY FORMULA Joins selector lets you specify the operating mode of the query by formula commands for the current process:<br>• 0: Uses the current settings of the database (default value). In databases created starting with version 11.2 of 4D v11 SQL, "SQL joins" are always activated for queries by formula. In converted databases, this mechanism is not activated by default for compatibility reasons but can be implemented via a preference.<br>• 1: Always use automatic relations (= functioning of previous versions of 4D). In this mode, a relation is necessary in order to set the selection of a table according to queries carried out on another table. 4D does not do "SQL joins."<br>• 2: Use SQL joins if possible (= default operation of databases created in version 11.2 and higher of 4D v11 SQL ). In this mode, 4D establishes "SQL joins" for queries by formula when the formula is suited for it (with two notable exceptions, see the description of the **QUERY BY FORMULA** or **QUERY SELECTION BY FORMULA** command).<br>**Note:** With 4D in remote mode, "SQL joins" can only be used if the formulas are executed on the server (they must have access to the records). To configure where formulas are to be executed, please refer to selectors 46 and 47. |
| HTTP compression level | Longint | 50 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| HTTP compression threshold | Longint | 51 | **Scope**:4D application<br>**Kept between two sessions**: No<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Server base process stack size | Longint | 53 | **Scope**: 4D Server<br>**Kept between two sessions**: No<br>**Possible values**: Positive longint.<br>**Description**: Size of the stack allocated to each preemptive system process on the server, expressed in bytes. The default size is determined by the system.<br>Preemptive system processes (processes of the 4D client base process type) are loaded to control the main 4D client processes. The size allocated by default to the stack of each preemptive process allows a good ease of execution but may prove to be consequential when very large numbers of processes (several hundred) are created. For optimization purposes, this size can be reduced considerably if the operations carried out by the database allow for it (for example if the database does not carry out sorts of large quantities of records). Values of 512 or even 256 KB are possible. Be careful, under-sizing the stack is critical and can be harmful to the operation of 4D Server. Setting this parameter should be done with caution and must take the database conditions of use into account (number of records, type of operations, etc.).<br>In order to be taken into account, this parameter must be executed on the server machine (for example in the **On Server Startup Database Method**). |
| Idle connections timeout | Longint | 54 | **Scope**:4D application unless value is negative<br>**Kept between two sessions**: No<br>**Possible values**: Whole value expressing a duration in seconds. The value can be positive (new connections) or negative (existing connections). By default, the value is 20.<br>**Description**: Maximum period of inactivity (timeout) for connections to both the 4D database engine and the SQL engine, as well as, in *ServerNet* mode (new network layer), to the 4D application server. When an idle connection reaches this limit, it is automatically put on standby, which freezes the client/server session and closes the network socket. In the server administration window, the state of the user process is indicated as "Postponed". This functioning is completely transparent for the user: as soon as there is new activity on the connection which is on standby, the socket is automatically reopened and the client/server session is restored.<br>On the one hand, this setting lets you save resources on the server: connections on standby close the socket and free up a process on the server. On the other hand, it lets you avoid losing connections due to the closing of idle sockets by the firewall. For this, the timeout value for idle connections must be lower than that of the firewall in this case.<br>If you pass a positive value in *value*, it applies to all new connections in all the processes. If you pass a negative value, it applies to connections that are open in the current process. If you pass 0, idle connections are not subjected to a timeout.<br>This parameter can be set on both the server and client side. If you pass two different durations, the shorter one is taken into account. Usually, you do not need to change this value. |
| PHP interpreter IP address | Longint | 55 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Formatted string of the type "nnn.nnn.nnn.nnn" (for example "127.0.0.1").<br>**Description**: IP address used locally by 4D to communicate with the PHP interpreter via FastCGI. By default, the value is "127.0.0.1". This address must correspond to the machine where 4D is located. This parameter can also be set globally for all the machines via the Database Settings.<br>For more information about the PHP interpreter, please refer to the *Design Reference* manual. |
| PHP interpreter port | Longint | 56 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Positive long integer type value. By default, the value is 8002.<br>**Description**: Number of the TCP port used by the PHP interpreter of 4D. This parameter can also be modified globally for all the machines via the Database Settings.<br>For more information about the PHP interpreter, please refer to the *Design Reference* manual. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| PHP number of children | Longint | 57 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Positive long integer type value. By default, the value is 5.<br>**Description**: Number of child processes to be created and maintained locally by the PHP interpreter of 4D. For optimization reasons, the PHP interpreter creates and uses a set (pool) of system processes called "child processes" for processing script execution requests. You can vary the number of child processes according to the needs of your application. This parameter can also be modified globally for all the machines via the Database Settings. For more information about the PHP interpreter, please refer to the *Design Reference* manual.<br>**Note:** Under Mac OS, all the child processes share the same port. Under Windows, each child process uses a specific port number. The first number is the one set for the PHP interpreter; the other child processes increment the first one. For example, if the default port is 8002 and you launch 5 child processes, they will use ports 8002 to 8006. |
| PHP max requests | Longint | 58 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Positive long integer type value. By default, the value is 500.<br>**Description**: Maximum number of requests accepted by the PHP interpreter. When this maximum number is reached, the interpreter returns errors of the "server busy" type. For security or performance reasons, you can modify this value. This parameter can also be modified globally for all the machines via the Database Settings. For more information about this parameter, please refer to the FastCGI-PHP documentation.<br>**Note:** On the 4D side, these parameters are applied dynamically; it is not necessary to exit 4D in order for them to be taken into account. On the other hand, if the PHP interpreter is already launched, it will be necessary to exit and relaunch it in order for these modifications to be taken into account. |
| PHP use external interpreter | Longint | 60 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values** : 0 = use internal interpreter, 1 = use external interpreter<br>**Description**: Value indicating whether PHP requests in 4D are sent to the internal interpreter provided by 4D or to an external interpreter. By default the value is 0 (use of interpreter provided by 4D). If you want to use your own PHP interpreter, for example in order to use additional modules or a specific configuration, pass 1 in *value*. In this case, 4D does not launch its internal interpreter in the case of PHP requests.<br>The custom PHP interpreter must have been compiled in FastCGI and be located on the same machine as the 4D engine. Note that in this case, you must manage the interpreter entirely; it will not be started nor stopped by 4D. This parameter can also be modified globally for all the machines via the Database Settings. |
| Maximum temporary memory size | Longint | 61 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Positive longint.<br>**Description**: Maximum size of temporary memory that 4D can allocate to each process, expressed in MB. By default, the value is 0 (no maximum size). 4D uses a special temporary memory dedicated to indexing and sorting operations. This memory is intended to preserve the "standard" cache memory during massive operations. It is activated only when needed. By default, the size of the temporary memory is limited only by the resources available (according to the system memory configuration).<br>This mechanism is suitable for most applications. However, in certain specific contexts, more particularly when a client-server application simultaneously carries out a large number of sequential sorts, the size of the temporary memory can increase critically, to the point where it can render the system unstable. In this context, setting a maximum size for the temporary memory allows you to preserve proper functioning of the application. In return, the running speed might be affected: when the maximum size is reached for a process, 4D uses disk files which may slow down processing. For specific needs such as those described above, a maximum size of around 50 MB is generally a good compromise. However, the ideal value will need to be determined according to the specificities of the application and will generally be the result of real-time volumetric testing. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| SSL cipher list | String | 64 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Sequence of strings separated by colons (for example "RC4-MD5:RC4-64-MD5:....")<br>**Description:** Cipher list used by 4D for the secure protocol. This list modifies the priority of ciphering algorithms implemented by 4D. For example, you can pass the following string in the *value* parameter: "AES:ALL:!aNULL:!eNULL:+RC4:@STRENGTH". For a complete description of the syntax for the ciphers list, refer to the ciphers page of the OpenSSL site.<br>This setting applies to the entire 4D application (it concerns the HTTP server, SQL server, client/server connections, as well as the HTTP client and all the 4D commands that make use of the secure protocol) but it is temporary (it is not maintained between sessions).<br>When the cipher list has been modified, you will need to restart the server concerned in order for the new settings to be taken into account.<br>To reset the cipher list to its default value (stored permanently in the SLI file), call the **SET DATABASE PARAMETER** command and pass an empty string ("") in the *value* parameter.<br>**Note:** With the **Get database parameter** command, the cipher list is returned in the optional *stringValue* parameter and the return parameter is always 0. |
| Cache unload minimum size | Longint | 66 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Positive longint > 1.<br>**Description**: Minimum size of memory to release from the database cache when the engine needs to make space in order to allocate an object to it (value in bytes).<br>The purpose of this selector is to reduce the number of times that data is released from the cache in order to obtain better performance. You can vary this setting according to the size of the cache and that of the blocks of data being handled in your database.<br>By default, if this selector is not used, 4D unloads at least 10% of the cache when space is needed. |
| Direct2D status | Longint | 69 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Description**: Activation mode to implement Direct2D under Windows.<br>**Possible values**: One of the following constants (mode 3 by default):<br>Direct2D Disabled (0): Direct2D mode is not enabled and the database functions in the previous mode (GDI/GDIPlus).<br>Direct2D Hardware (1): Use Direct2D as graphics hardware context for entire 4D application. If this context is not available, use Direct2D graphics software context (except under Vista, in which case GDI/GDIPlus mode is used for better performance).<br>Direct2D Software (3) (Default mode): Beginning with Windows 7, use Direct2D graphics software context for entire 4D application. Under Vista, GDI/GDIPlus mode is used for better performance.<br>**Compatibility note:** Starting with 4D v14, hybrid modes are disabled and redirected to available modes (the former mode 2 is equivalent to 1; former modes 4 and 5 are equivalent to mode 3). |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Direct2D get active status | Longint | 74 | **Note**: You can only use this selector with the **Get database parameter** command and its value cannot be set.<br>**Description**: Returns active implementation of Direct2D under Windows.<br>**Possible values**: 0, 1, 2, 3, 4 or 5 (see values of selector 69). The value returned depends on the availability of Direct2D, the hardware and the quality of Direct2D support by the operating system.<br>For example, if you execute:<br>`SET DATABASE PARAMETER(Direct2D status;Direct2D Hardware)`<br>`$mode:=Get database parameter(Direct2D get active status)`<br><br>- On Windows 7 and higher, *$mode* is set to 1 when the system detects hardware compatible with Direct2D; otherwise, *$mode* is set to 3 (software context).<br>- On Windows Vista, *$mode* is set to 1 when the system detects hardware compatible with Direct2D; otherwise, *$mode* is set to 0 (disabling of Direct2D).<br>- On Windows XP, *$mode* is always set to 0 (not compatible with Direct2D). |
| Diagnostic log recording | Longint | 79 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: 0 or 1 (0 = do not record, 1 = record)<br>**Description**: Starts or stops recording of the 4D diagnostic file. By default, the value is 0 (do not record).<br>4D can continuously record a set of events related to the internal application operation into a diagnostic file. Information contained in this file is intended for the development of 4D applications and can be analyzed with the help of the 4D tech support. When you pass 1 in this selector, a diagnostic file, named *DatabaseName_X*.txt, is automatically created (or opened) in the database **Logs** folder. Once this file reaches a size of 10 MB, it is closed and a new file named *DatabaseName_X*.txt is generated, with an incremented sequence number X.<br>Note that you can include custom information in this file using the **LOG EVENT** command. |
| Log command list | String | 80 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: String containing a list of 4D command numbers to record (separated by semi-colons) or "all" to record all the commands or "" (empty string) to record none of them.<br>**Description**: List of 4D commands to record in the debugging file (see selector 34, Debug Log Recording). By default, all 4D commands are recorded.<br>This selector restricts the quantity of information saved in the debugging file by limiting the 4D commands whose execution you want to record. For example, you can write:<br>`SET DATABASE PARAMETER(Log command list;"277;341") //Record only the QUERY and QUERY SELECTION commands` |
| Spellchecker | Longint | 81 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: 0 (default) = native OS X spellchecker (Hunspell disabled), 1 = Hunspell spellcheck enabled.<br>**Description**: Enables the Hunspell spellcheck under OS X. By default, the native spellchecker is enabled on this platform. You may prefer to use the Hunspell spellcheck, for example, in order to unify the interface for your cross-platform applications (under Windows, only the Hunspell spellcheck is available). For more information, refer to **Support of Hunspell dictionaries**. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| QuickTime support | Longint | 82 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (default) = QuickTime disabled, 1 = QuickTime enabled.<br>**Description**: In 4D starting with v14, by default QuickTime codecs are no longer supported. For compatibility, you can use this selector to re-enable them in your database. Modification of this option requires that the database be restarted. Nevertheless, you should note that in future versions of 4D, QuickTime support is permanently removed. |
| JSON use local time | Longint | 85 | **Scope**: Current process<br>**Kept between two sessions**: No<br>**Possible values**: 0 = ignore local time zone, 1 (default) = take time zone into account.<br>**Description**: By default, 4D dates converted to JSON format take the local time zone into account. For example, converting the date !23/08/2013! gives you "2013-08-22T22:00:00Z" in JSON format when the operation is performed in France during Daylight Savings Time (GMT+2). This principle conforms to the standard operation of JavaScript.<br>This can be a source of errors when you want to send JSON date values to someone in a different time zone. This is the case, for example, when you export a table using **Selection to JSON** in France that is meant to be reimported in the US using **JSON TO SELECTION**. By default, since dates are re-interpreted in each time zone, the values stored in the database will be different. In this case, you can modify the conversion mode for dates so that they do not take the time zone into account by passing 0 in this selector. Converting the date !23/08/2013! will then give you "2013-08-23T00:00:00Z" in all cases. |
| Use legacy network layer | Longint | 87 | **Scope**: 4D in local mode, 4D Server<br>**Kept between two sessions**: Yes<br>**Description:** Sets or gets the current status of the legacy network layer for client/server connections. The legacy network layer is obsolete beginning with 4D v14 R5 and should be replaced progressively in your applications with the *ServerNet* network layer. *ServerNet* will be required in upcoming 4D releases in order to benefit from future network evolutions. For compatibility reasons, the legacy network layer is still supported to allow a smooth transition for existing applications; (it is used by default in applications converted from a release prior to v14 R5). Pass 1 in this parameter to use the legacy network layer (and disable *ServerNet*) for your client/server connections, and pass 0 to disable the legacy network (and use the *ServerNet*).<br>This property can also be set by means of the "Use legacy network layer" option found on the **Compatibility page** of the Database Settings (see **Network and Client-Server options**). In this section, you will also find a discussion about migration strategy. We recommend that you activate the *ServerNet* as soon as possible.<br>You will need to restart the application in order for this parameter to be taken into account. It is not available in 4D Server v14 R5 64-bit version for OS X, which only supports the *ServetNet*; (it always returns 0).<br>**Possible values:** 0 or 1 (0 = do not use legacy layer, 1 = use legacy layer)<br>**Default value:** 0 in databases created with 4D v14 R5 or higher, 1 in databases converted from 4D v14 R4 or earlier. |
| SQL Server Port ID | Longint | 88 | **Scope**: 4D local, 4D Server.<br>**Kept between two sessions**: Yes<br>**Description**: Gets or sets the TCP port number used by the integrated SQL server of 4D in local mode or 4D Server. By default, the value is 19812. When this selector is set, the database setting is updated. You can also set the TCP port number on the "SQL" page of the Database Settings dialog box.<br>**Possible values**: 0 to 65535.<br>**Default value**: 19812 |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Circular log limitation | Longint | 90 | **Scope**: 4D local, 4D Server.<br>**Kept between two sessions**: No<br>**Possible values**: Any integer value, 0 = keep all logs<br>**Description**: Maximum number of files to keep in rotation for each type of log. By default, all files are kept. If you pass a value *X*, only the *X* most recent files are kept, with the oldest being erased automatically when a new one is created. This setting applies to each of the following log files: request logs (selectors 28 and 45), debug log (selector 34), events log (selector 79), as well as Web request logs and Web debug logs (selectors 29 and 84 of the **WEB SET OPTION** command). |
| Number of formulas in cache | Longint | 92 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Positive longints<br>**Default value**: 0 (no cache)<br>**Description**: Sets or gets the maximum number of formulas to be kept in the cache of formulas, which is used by the **EXECUTE FORMULA** command. This limit is applied to all processes, but each process has its own formula cache. Caching formulas accelerates the **EXECUTE FORMULA** command execution in compiled mode since each cached formula is tokenized only once in this case. When you change the cache value, existing contents are reset even if the new size is larger than the previous one. Once the maximum number of formulas in the cache is reached, a new executed formula will erase the oldest one in the cache (FIFO mode). This parameter is only taken into account in compiled databases or compiled components. |
| Cache flush periodicity | Longint | 95 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: No<br>**Possible values**: longint > 1 (seconds)<br>**Description**: Gets or sets the current cache flush periodicity, expressed in seconds. Modifying this value overrides the **Flush Cache every X Seconds** option in the **Database/Memory page** of the Database settings for the session (it is not stored in the Database settings). |

## Example 1

The following method allows you to get 4D scheduler current values:

```
C_LONGINT($ticksbtwcalls;$maxticks;$minticks;$lparams)
If(Application type=4D Local Mode) ` 4D local mode is used
   $lparams:=Get database parameter(4D Local Mode Scheduler)
   $ticksbtwcalls:=$lparams &0x00ff
   $maxticks:=($lparams>>8)&0x00ff
   $minticks:=($lparams>>16)&0x00ff
End if
```

## Example 2

The selector 16 (IP address to listen) lets you get the IP address on which the 4D Web server receives HTTP requests. The following example splits up the hexadecimal value:

```
C_LONGINT($a;$b;$c;$d)
C_LONGINT($addr)
$addr:=Get database parameter(IP Address to listen)
$a:=($addr>>24)&0x000000ff
$b:=($addr>>16)&0x000000ff
$c:=($addr>>8)&0x000000ff
$d:=$addr&0x000000ff
```

# Get last update log path

Get last update log path -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Text |  | Pathname of most recent update log |

## Description

The **Get last update log path** command returns the complete pathname of the most recent update log file found on the machine where it is called.

The update log is generated by 4D during automatic update processes. It contains information about the updates performed as well as any errors that occurred.

This command is intended to be used in an automatic update process for a merged application (server or single-user). For more information, refer to **Finalizing and deploying final applications** in the *Design Reference* manual.

# GET SERIAL INFORMATION

GET SERIAL INFORMATION ( key ; user ; company ; connected ; maxUser )

| Parameter | Type | | Description |
|---|---|---|---|
| key | Longint | ⇦ | Unique product key (encrypted) |
| user | String | ⇦ | Registered name |
| company | String | ⇦ | Registered organization |
| connected | Longint | ⇦ | Number of connected users |
| maxUser | Longint | ⇦ | Maximum number of connected users |

## Description

The **GET SERIAL INFORMATION** command returns various information about the 4D current version serialization.

- *key*: unique ID of the installed product. A unique number is associated to a 4D application (such as 4D Server, 4D in local mode, 4D Desktop, etc.) installed on a machine. This number is encrypted, of course.
- *user*: Name application user as defined when installing.
- *company*: User's company or organization name as defined during installation.
- *connected*: Number of connected users when executing the command.
- *maxUsers*: Maximal number of users concurrently connected.

**Note:** The last two parameters always return 1 for 4D single user except in demonstration versions (0 is then returned).

**GET SERIAL INFORMATION** is part of the general component protection scheme implemented in 4D. Component developers can associate a copy of their product to a given installed 4D application, in order to avoid any illegal copies.

The serialization works as follows: a user who wants to get a component sends his unique key generated through the **GET SERIAL INFORMATION** command to the developer. This can be done through an Order form included in a demo version of the component. The generated key is unique and is associated to a specific 4D application.

The component developer can then generate his own serial number combining the key and a given cipher. The delivered component will offer a function verifying if the information returned by the **GET SERIAL INFORMATION** matches this serial number. Otherwise, the user will not be able to use the component.

**Note:** Plug-ins developers can use this protection scheme too. For more information, refer to the *4D Plugin API Reference*.

# Get table fragmentation

Get table fragmentation ( aTable ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to get the fragmentation rate |
| Function result | Real | ⇐ | Percentage of fragmentation |

## Description

The **Get table fragmentation** command returns the percentage of logical fragmentation for the records of the table designated by the *aTable* parameter.

The rate of logical fragmentation of the records indicates whether the records are stored in an ordered manner in the data file. If the fragmentation becomes too high, this can considerably slow down sorts and sequential searches on the table. A fragmentation percentage of 0 corresponds to no fragmentation. Beyond a rate of 20%, it may be useful to compact the data file.

## Example

This maintenance method lets you request the compacting of the data file in the case where there is considerable fragmentation in at least one table of the database:

```
ToBeCompacted:=False
For($i ;1;Get last table number)
   If(Is table number valid($i))
      If(Get table fragmentation(Table($i)->)>20)
         ToBeCompacted:=True
      End if
   End if
End for
If(ToBeCompacted)
 // Places a marker requesting compacting
End if
```

# Is compiled mode

Is compiled mode {( * )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | Returns information about host database |
| Function result | Boolean | ⮌ | Compiled (True), Interpreted (False) |

## Description

**Is compiled mode** tests whether you are running in compiled mode (True) or interpreted mode (False).

The optional * parameter is useful in the case of an architecture using components: it can be used to determine the database (host or component) for which you want to find out the running mode.

- When the command is called from a component:
  - If the * parameter is passed, the command returns **True** or **False** depending on the mode in which the host database is running,
  - If the * parameter is not passed, the command returns **True** or **False** depending on the mode in which the component is running.
- When the command is called from a method of the host database, it returns **True** or **False** depending on the mode in which the host database is running.

## Example

In one of your routines, you include debugging code useful only when you are running in interpreted mode, so surround this debugging code with a test that calls **Is compiled mode**:

```
  ` ...
If(Not(Is compiled mode))
  ` Include debugging code here
End if
  ` ...
```

# Is data file locked

Is data file locked -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Boolean | True = file/segment locked False = file/segment not locked |

## Description

The **Is data file locked** command returns True if the data file of the open database or at least one of its segments is locked — i.e. write protected.

Placed, for instance, in the **On Startup database method**, this command enables the prevention of any risk of accidental opening of a locked data file.

## Example

This method will prevent the opening of the database if the data file is locked:

```
If(Is data file locked)
   ALERT("The data file is locked. Impossible to open database.")
   QUIT 4D
End if
```

# NOTIFY RESOURCES FOLDER MODIFICATION

NOTIFY RESOURCES FOLDER MODIFICATION
Does not require any parameters

## Description

The **NOTIFY RESOURCES FOLDER MODIFICATION** command "forces" 4D Server to send a notification to all the connected 4D machines indicating that the *Resources* folder of the database has been modified so that they can synchronize their local *Resources* folder.

This command can be used more particularly to manage the synchronization of the *Resources* folders of remote machines after this folder has been modified by a stored procedure on the server.

For more information about managing the *Resources* folder in remote mode, please refer to the 4D Server Reference Guide.

Only the information that modification has occurred is sent. Remote machines react according to current settings. The options are the following:

- No synchronization of the local *Resources* local folder during the session,
- Automatic synchronization of the local *Resources* folder during the session,
- Display of a warning so that the user may carry out a synchronization if desired.

Current settings are set either:

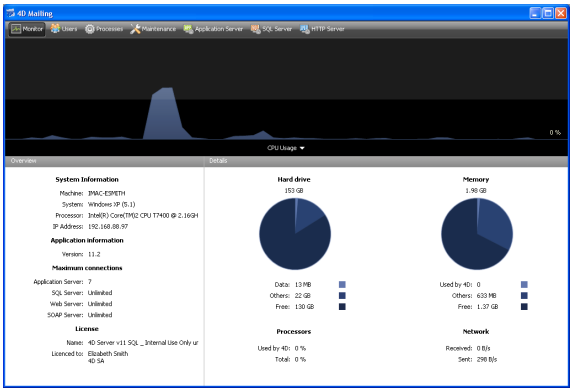
- at the overall database level using the **Update Resources folder during a session** parameter of the Database settings. In this case, it applies to all the remote machines;
- locally, using the **SET DATABASE PARAMETER** command executed on the remote machine (Auto synchro resources folder selector). In this case, it "overrides" the database setting and applies only to the remote machine for the duration of the session.

# OPEN ADMINISTRATION WINDOW

OPEN ADMINISTRATION WINDOW
Does not require any parameters

## Description

The **OPEN ADMINISTRATION WINDOW** command displays the server administration window on the machine that executes it. The 4D Server administration window can be used to view the current parameters and to carry out various maintenance operations (see the 4D Server Reference Guide). Beginning with version 11 of 4D Server, this window can be displayed from a client machine:




This command must be called in the context of a connected 4D application or a 4D Server. It does nothing if:

- it is called in a 4D application in local mode,
- it is executed by a user other than the Designer or the Administrator (in this case, the error -9991 is generated, see the **Database Engine Errors (-10602 -> 4004)** section).

## Example

Here is the code for an administration button:

```
If(Application type=4D local mode)
   OPEN SECURITY CENTER
   ...
End if
If(Application type=4D remote mode)
   OPEN ADMINISTRATION WINDOW
   ...
End if
If(Application type=4D Server)
   ...
   OPEN SECURITY CENTER
End if
```

## System variables and sets

If the command has been executed correctly, the OK system variable is set to 1. Otherwise, it is set to 0.

# OPEN DATA FILE

OPEN DATA FILE ( accessPath )

| Parameter | Type | | Description |
|---|---|---|---|
| accessPath | String | ⇨ | Name or complete access path of the data file to open |

## Description

The **OPEN DATA FILE** command allows changing the data file opened by the 4D application on-the-fly.

Pass the name or the full access path of the data file to open (file with a ".4DD" suffix) in the *accessPath* parameter. If you pass only the file name, it must be placed next to the structure file of the database.

If the access path sets a valid data file, 4D quits the database in progress and re-opens it with the specified data file. In single-user mode, the **On Exit database method** and the **On Startup database method** are successively called.

**Warning:** Since this command causes the application to quit before re-opening with the specified data file, it must be used with precaution in the **On Startup database method** or in a method called by this database method, so as to avoid generating an infinite loop.

The command is executed in an asynchronous manner: after its call, 4D continues executing the rest of the method. Then, the application behaves as if the **Quit** command was selected in the **File** menu: open dialog boxes are cancelled, any open processes have 10 seconds to finish before being terminated, etc.

Before launching the operation, the command checks the validity of the specified data file. Also, if the file was already open, the command verifies that it corresponds to the current structure.

If you pass an empty string in *accessPath*, the command will re-open the database without changing the data file.

**4D Server:** Beginning with 4D v13, this command can be executed with 4D Server. In this context, it makes an internal call to **QUIT 4D** on the server (which causes a dialog box to appear on each remote machine indicating that the server is in the process of quitting) before opening the designated file.

## Example

In the context of deploying a merged application, you want to open or create the user data file in the On Startup database method. This example uses the default data file (see **Data file management in final applications**):

```
If(Data file="@default.4dd")
   If(Version type?? Merged application)
      If(Is data file locked)
         $dataPath:=Get 4D folder(Active 4D Folder)+"data.4dd"
 //If a local data file already exists
         If(Test path name($dataPath)=Is a document)
            OPEN DATA FILE($dataPath) //open it
         Else
            CREATE DATA FILE($dataPath) //create it
         End if
      End if
   End if
End if
```

# OPEN DATABASE

OPEN DATABASE ( filePath )

| Parameter | Type | | Description |
|---|---|---|---|
| filePath | String | ⇒ | File name (.4db, .4dc, .4dbase or .4dlink) or complete access path of database to open |

## Description

The **OPEN DATABASE** command closes the current 4D database and opens on-the-fly the database defined by *filePath*. This command is useful for automatic testing purposes, or to reopen a database automatically after a compilation.

In the *filePath* parameter, pass the name or full access path of the database to be opened. You can use files having one of the following extensions:

- .4db (interpreted structure file),
- .4dc (compiled structure file),
- .4dbase (OS X package),
- .4dlink (shortcut file).

If you pass only a file name, this file must be placed at the same level as the structure file of the current database.

If the access path defines a valid database, 4D quits the database that is already open and then opens the specified database. In single-user mode, the **On Exit database method** of the database being closed and the **On Startup database method** of the database being opened are called successively.

**Warning:** Since this command causes the application to quit before re-opening with the specified database, it is not recommended to use it in the **On Startup database method**, or in a method called by this database method.

The command is executed in an asynchronous manner: after its call, 4D continues executing the rest of the method. Then, the application behaves as if the **Quit** command was selected in the **File** menu: open dialog boxes are cancelled, any open processes have 10 seconds to finish before being terminated, and so on.

If the target database file is not found or is invalid, a standard file system error is returned and 4D does nothing.

This command can only be executed from a standard database. If it is called from an engined application (single user or server), the error -10509 "Can't open database" is returned.

## Example

```
OPEN DATABASE("C:¥¥databases¥¥Invoices¥¥Invoices.4db")
```

# OPEN SECURITY CENTER

OPEN SECURITY CENTER
Does not require any parameters

## Description

The **OPEN SECURITY CENTER** command displays the Maintenance and Security Center (MSC) window.

Depending on the access rights of the current user, some functions available in this window may be disabled.

**Note:** This command works on the same principle as a call to **DIALOG** with the * parameter: the MSC is displayed in a window and the command immediately returns control to the 4D code. If the current process finishes, the window is automatically closed by simulating a **CANCEL**. So you need to manage its display through the code of the process being executed.

# OPEN SETTINGS WINDOW

OPEN SETTINGS WINDOW ( selector {; access}{; settingsType} )

| Parameter | Type | | Description |
|---|---|---|---|
| selector | String | ⇒ | Key designating a theme or a page or a group of parameters in the Preferences or Settings dialog box |
| access | Boolean | ⇒ | True=Lock the other pages of the dialog box False or omitted=Leave the other pages of the dialog box active |
| settingsType | Longint | ⇒ | 0 or omitted = Structure settings, 1 = User settings |

## Description

The **OPEN SETTINGS WINDOW** command opens the Preferences dialog box of 4D or the Database Settings of the current 4D application and displays the parameters or the page corresponding to the key passed in *selector*.

The *selector* parameter must contain a “key” indicating the dialog box and the page to opened. This key is constructed as follows: */Dialog{/Page{/Parameters}}*. *Dialog* indicates the dialog box to be displayed: you can pass "4D" (for the Preferences) or "Database" (for Database Settings). For example, to indicate the Compiler page of the Database Settings, *selector* should contain "*/Database/Compiler*". The list of keys that can be used is provided below. If you just pass a slash ("/") in *selector*, the command displays the first page of the Database Settings dialog box.

The *access* parameter lets you control user actions in the Preferences or Database Settings dialog box by locking the other pages. Typically, you may want for the user to be able to customize certain parameters while preventing others from being modified. In this case, passing True in the *access* parameter means that only the page specified by the *selector* parameter will be active and modifiable, while access to all other pages will be locked (clicking on the buttons in the navigation bar will have no effect). If you pass False or omit the *access* parameter, all the pages of the dialog box will be accessible with no restriction.

The *settingsType* parameter is taken into account in databases configured in "User settings" mode only (in this mode, custom "User settings" or "User settings for data file" are generated in an external file and used instead of the standard settings, see the **User settings** section in the *Design Reference* manual). In this context, this parameter lets you indicate whether you want to access the "Structure settings", the "User settings", or the "User settings for data file" dialog box. You pass one of the following constants, found in the "**4D Environment**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Structure settings | Longint | 0 | Access to "Structure settings" (default value if parameter omitted). In this mode, values used for *selector* are identical to those in standard mode. |
| User settings | Longint | 1 | Access to "User settings". In this mode, only certain keys can be used in the *selector* parameter |
| User settings for data | Longint | 2 | Access to "User settings for data file", that is, user settings stored at the same level as the data file. In this mode, only certain keys can be used with the *selector* parameter (same subset as the User settings) |

If you pass an invalid key, the first page of the Database Settings dialog box is displayed.

### Path keys (standard mode)

Here are the keys that can be used in the *selector* parameter in standard mode, in other words with the "Structure settings":

*/4D*
*/4D/General*
*/4D/Structure*
*/4D/Form editor*
*/4D/Method editor*
*/4D/Client-Server*
*/4D/Shortcuts*
*/Database*
*/Database/General*
*/Database/Mover*
*/Database/Interface*
*/Database/Interface/Developper*
*/Database/Interface/User*
*/Database/Interface/Shortcuts*

*/Database/Compiler*
*/Database/Database*
*/Database/Database/Data storage*
*/Database/Database/Memory and cpu*
*/Database/Database/International*
*/Database/Backup*
*/Database/Backup/Scheduler*
*/Database/Backup/Configuration*
*/Database/Backup/Backup and restore*
*/Database/Client-Server*
*/Database/Client-Server/Network*
*/Database/Client-Server/IP configuration*
*/Database/Web*
*/Database/Web/Config*
*/Database/Web/Options 1*
*/Database/Web/Options 2*
*/Database/Web/Log format*
*/Database/Web/Log scheduler*
*/Database/Web/Webservices*
*/Database/SQL*
*/Database/php*
*/Database/Compatibility*
*/Database/Security*

**Compatibility note:** You can still use keys defined for 4D versions 11.x or previous using this command; 4D automatically establishes the correspondence. However, we recommend that you replace the former calls with the keys listed above.

### Path keys (User settings mode)

Here are the keys that can be used in the *selector* parameter in "User settings" mode:

*/Database*
*/Database/Interface*
*/Database/Database/Memory and cpu*
*/Database/Client-Server*
*/Database/Client-Server/Network*
*/Database/Client-Server/IP configuration*
*/Database/Web*
*/Database/Web/Config*
*/Database/Web/Options 1*
*/Database/Web/Options 2*
*/Database/Web/Log format*
*/Database/Web/Log scheduler*
*/Database/Web/Webservices*
*/Database/SQL*
*/Database/php*

## Example 1

Open the “Methods” page of the 4D Preferences:

```
OPEN SETTINGS WINDOW("/4D/Method editor")
```

## Example 2

Open the “Shortcuts” parameters in the Database Settings while locking the other settings:

```
OPEN SETTINGS WINDOW("/Database/Interface/Shortcuts";True)
```

## Example 3

Open Database Settings on the first page:

```
OPEN SETTINGS WINDOW("/")
```

## Example 4

Access to the Interface page of the Database settings in "User settings" mode:

```
OPEN SETTINGS WINDOW("/Database/Interface";1)
```

## System variables and sets

If the Preferences/Settings dialog box is validated, the system variable OK returns 1. Otherwise, it returns 0.

# PLUGIN LIST

PLUGIN LIST ( numbersArray ; namesArray )

| Parameter | Type | | Description |
|---|---|---|---|
| numbersArray | Longint array | ⇦ | Numbers of plug-ins |
| namesArray | String array | ⇦ | Names of plug-ins |

## Description

The **PLUGIN LIST** command fills in the *numbersArray* and *namesArray* arrays with the numbers and names of the plug-ins loaded and usable by the 4D application. These two arrays are automatically sized and synchronized by the command.

**Note:** You can compare the values returned in *numbersArray* with the constants of the **Is License Available** theme.

**PLUGIN LIST** takes all plug-ins into account, including those that are integrated (for example, 4D Chart), and third-party plug-ins.

# QUIT 4D

QUIT 4D {( time )}

| Parameter | Type | | Description |
|---|---|---|---|
| time | Longint | ⇨ | Time (sec) before quitting the server |

## Description

The **QUIT 4D** command exits the current 4D application and returns to the Desktop.

The command processing is different whether it is executed on 4D (local or remote mode) or on 4D Server.

### With 4D local mode and remote mode

After you call **QUIT 4D**, the current process stops its execution, then 4D acts as follows:

- If there is an **On Exit database method**, 4D starts executing this method within a newly created local process. For example, you can use this database method to inform other processes, via interprocess communication, that they must close (data entry) or stop the execution of operations started by the **On Startup database method** (connection from 4D to another database server). Note that 4D will eventually quit; the **On Exit database method** can perform all the cleanup or closing operations you wish, but cannot refuse the quit and will at some point end.
- If there is no **On Exit database method**, 4D aborts each running process one by one, without distinction.

If the user is performing data entry, the records will be cancelled and not saved.

If you want to let the user save data entry modifications made in the current open windows, you can use interprocess communication to signal all the other user processes that the database is going to be exited. To do so, you can adopt two strategies:

- Perform these operations from within the current process before calling **QUIT 4D**
- Handle these operations from within the **On Exit database method**.

A third strategy is also possible. Before calling **QUIT 4D**, you check whether a window will need validation; if that is the case, you ask the user to validate or cancel these windows and then to choose Quit again. However, from a user interface standpoint, the first two strategies are preferable.

**Note:** The *time* parameter cannot be used with 4D in local or remote mode.

### With 4D Server (Stored procedure)

The **QUIT 4D** command can be executed on the server machine, in a stored procedure. In this case, it accepts the *time* optional parameter.

The *time* parameter allows setting a timeout to the 4D Server before the application actually quits, allowing client machines the time to disconnect. You must pass a value in seconds in *time*. This parameter is only taken into consideration during an execution on the server machine. With 4D in local or remote mode, it is ignored.

If you do not pass a parameter for *time*, 4D Server will wait until all client machines are disconnected before quitting.

Unlike 4D, the processing of **QUIT 4D** by 4D Server is asynchronous: the method where the command is called is not interrupted after is has been executed.

If there is an **On Server Shutdown Database Method**, it is executed after the delay set by the *time* parameter, or after all clients have disconnected, depending on your parameters.

The action of the **QUIT 4D** command used in a stored procedure is the same as the one for the Quit command of the 4D Server File menu: it causes a dialog box to appear on each client machine indicating that the server is about to quit.

## Example

The project method listed here is associated with the Quit or Exit menu item in the File menu.

```
` M_FILE_QUIT Project Method

CONFIRM("Are you sure that you want to quit?")
```

```
If(OK=1)
   QUIT 4D
End if
```

# RESTART 4D

RESTART 4D {( time {; message} )}

| Parameter | Type | | Description |
|---|---|---|---|
| time | Longint | ⇒ | Time delay (seconds) before 4D restarts |
| message | String | ⇒ | Text to display on client machines |

## Description

The **RESTART 4D** command restarts the current 4D application.

This command is mainly intended for use in the context of a merged application (client/server or single-user) and in conjunction with the **SET UPDATE FOLDER** command. In this case, the automatic update process is launched: the new version of the application designated by **SET UPDATE FOLDER** automatically replaces the current version at the time of the restart resulting from **RESTART 4D**. The pathname of the data file is saved and used automatically.

If no update information was specified using the **SET UPDATE FOLDER** command in the current session, the command simply restarts the 4D application with the current structure and data files.

You can use the *time* parameter to defer restarting the application in order to give client machines time to disconnect. You must pass a value in seconds for the *time*. If you omit this parameter, the server application waits, for a maximum of 10 minutes, for all the client applications to be disconnected. After that, all client applications are automatically disconnected.

**Note:** The *time* and *message* parameters are only taken into account with server applications (they are ignored if the command is executed in a single-user or remote application).

The optional *message* parameter displays a custom message for connected client applications.

If the command is executed correctly, the OK system variable is set to 1; otherwise, it is set to 0 and the application restarts. You can intercept any errors generated by the command using a method installed using the **ON ERR CALL** command.

# SET DATABASE LOCALIZATION

SET DATABASE LOCALIZATION ( languageCode {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| languageCode | Text | ➡ | Language selector |
| * | Operator | ➡ | Scope of command |

## Description

The **SET DATABASE LOCALIZATION** command is used to modify the current language of the database for the current session.

The current language of the database lets you specify the .lproj folder where the program will look for the localized elements of the application (text and pictures). By default, 4D automatically determines the current language according to the contents of the **Resources** folder and the system environment (see the description of the **Get database localization** command). **SET DATABASE LOCALIZATION** can be used to modify the default current language.

The command does not modify the language of forms that are already loaded, only elements displayed after the command is called will take the new configuration into account.

Pass the language to be used for the application in *languageCode*, expressed in the standard specified by RFC 3066, ISO639 and ISO3166. Typically, you must pass "fr" for French, "es" for Spanish, "en-us" for American English, and so on. For more information about this standard, please refer to **MissingRef** in the *Design Reference* manual.

By default, the command applies to all the databases and components that are open, for all the processes. The optional * parameter, if passed, specifies that the command must only apply to the database that called it. This parameter can be used more particularly to specify separately the language of the database and that of a component.

If the command has been executed correctly, the *OK* system variable is set to 1; otherwise, it is set to 0.

**Note:** In accordance with the RFC, the command uses the "-" (dash) to separate the language code and the region code, for example "fr-ca" or "en-us". On the other hand, the "lproj" folders of the **Resources** folder use the "_" (underscore), for example "fr_ca.lproj" or "en_us.lproj".

**4D Server:** With 4D Server, the languages available are those located on the remote machine that called the command. You must therefore make sure that the **Resources** folders are synchronized.

## Example 1

We want to set French as the interface language:

```
SET DATABASE LOCALIZATION("fr")
```

## Example 2

The interface of your application uses the static string ":xliff:shopping". The XLIFF files contain more particularly the following information:

- FR folder:

```
<trans-unit id="15" resname="Shopping"> <source>Shopping</source> <target>Faire les courses</target> </trans-unit>
```

- FR_CA folder:

```
<trans-unit id="15" resname="Shopping"> <source>Shopping</source> <target>Magasiner</target> </trans-unit>
```

```
SET DATABASE LOCALIZATION("fr")
 //the string ":xliff:shopping" displays "Faire les courses"
```

```
SET DATABASE LOCALIZATION("fr-ca")
 //the string ":xliff:shopping" displays "Magasiner"
```

# SET DATABASE PARAMETER

SET DATABASE PARAMETER ( {aTable ;} selector ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to set the parameter or, Default table if this parameter is omitted |
| selector | Longint | ⇒ | Code of the database parameter to modify |
| value | Real, String | ⇒ | Value of the parameter |

## Description

The **SET DATABASE PARAMETER** command allows you to modify various internal parameters of the 4D database.

The *selector* designates the database parameter to modify. 4D offers predefined constants, which are located in the "**Database Parameters**" theme. The following table lists each constant, describes its scope and indicates whether any changes made are kept between two sessions:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Direct2D disabled | Longint | 0 | See selector 69 (Direct2D Status) |
| Direct2D hardware | Longint | 1 | See selector 69 (Direct2D Status) |
| Direct2D software | Longint | 3 | See selector 69 (Direct2D Status) |
| Minimum Web process | Longint | 6 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 -> 32 767<br>**Description**: Minimum number of Web processes to maintain in non-contextual mode with 4D in local mode and 4D Server. By default, the value is 0 (see below). |
| Maximum Web process | Longint | 7 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 -> 32 767<br>**Description**: Maximum number of Web processes to maintain in non-contextual mode with 4D in local mode and 4D Server. By default, the value is 10.<br>In non-contextual mode, for the Web server to be reactive, 4D delays the Web processes for 5 seconds and reuses them to execute any possible future HTTP queries. In terms of performance, this is actually more advantageous than creating a new process for each query. Once a Web process is reused, it is delayed again for 5 seconds. When the maximum number of Web processes has been reached, the web process is then aborted. If no query has been attributed to a Web process within the 5 second delay, the process is aborted, except if the minimum number of Web processes has been reached (in which case the process is delayed again).<br>These parameters allow you to adjust how your Web server operates in relation to the number of requests and the memory available as well as other parameters. |
| _o_Web conversion mode | Longint | 8 | **** *Selector disabled* **** |
| _o_Database cache size | Longint | 9 | **Scope**: 4D application<br>**Kept between two sessions**: -<br>**Description**: *Constant obsolete (kept for compatibility reasons only)*. We now recommend using the **Get cache size** command. |
| 4D Local mode scheduler | Longint | 10 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Description**: see selector 12 |
| 4D Server scheduler | Longint | 11 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Description**: see selector 12 |

| Constant | Type | Value | Comment |
|---|---|---|---|
| 4D Remote mode scheduler | Longint | 12 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Possible values**: for selectors 10, 11 and 12, the *value* parameter is expressed in hexadecimal *0x00aabbcc* as follows:<br>*aa* = minimum number of ticks per call to the system (0 to 100 included).<br>*bb* = maximum number of ticks per call to the system (0 to 100 included).<br>*cc* = number of ticks between calls to the system (0 to 20 included).<br>If one of the values is out of range, 4D sets it to its maximum. You can pass one of the following preset standard values in the *value* parameter:<br>• *value* = -1: maximum priority allocated to 4D,<br>• *value* = -2: average priority allocated to 4D,<br>• *value* = -3: minimum priority allocated to 4D.<br>**Description**: This parameter allows you to dynamically set the 4D system internal calls. Depending on the Selector, the scheduler value will be set for:<br>• 4D local mode when the command is called from a 4D single-user application (*selector=10*).<br>• 4D Server when the command is called from 4D Server (*selector=11*).<br>• 4D remote mode when the command is called from a 4D connected to 4D Server (*selector=12*).<br>**Note:** The operation of selector 12 (4D Remote Mode Scheduler) differs according to whether the **SET DATABASE PARAMETER** command is executed on the server machine or on the client machine:<br>- If the command is executed on the server machine, the new value will be applied to all the client machines that connect to it subsequently.<br>- If the command is executed on the client machine, the new value is applied to the client machine immediately as well as to all the client machines that connect to the server subsequently.<br>You can use this operation to implement a dynamic and individualized management of priority for each client machine. This consists in executing the command initially on the client machine to be configured, then a second time on the server machine using the default value, which will then be used for the client machines that connect to it subsequently.<br>This operation is in effect in 4D starting with versions 6.8.6, 2003.3 and 2004.<br>**Warning**: Configuring these selectors inappropriately can cause serious degradation of application performance. It is recommended to only modify the default values with full knowledge of the facts. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| 4D Server timeout | Longint | 13 | **Scope**: 4D application if *value* positive<br>**Kept between two sessions**: Yes if *value* positive<br>**Possible values**: 0 -> 32 767<br>**Description**: Value of the 4D Server timeout. The default 4D Server timeout value is defined on the "Client-Server/Network options" page of the Database settings dialog box on the server side.<br>The server timeout sets the maximum period "authorized" to wait for a client response, for example when it is executing a blocking operation. After this period, 4D Server disconnects the client. The 4D Server Timeout selector allows you to set, in the corresponding *value* parameter, a new timeout expressed in minutes. This feature is particularly useful to increase the timeout before executing a blocking and time-consuming operation on the client, such as printing a large number of pages, which can cause an unexpected timeout.<br>You also have two options:<br>• If you pass a **positive** value in the *value* parameter, you set a global and permanent timeout: the new value is applied to all process and is stored in the preferences of the 4D application (equivalent to change in the Preferences dialog box).<br>• If you pass a **negative** value in the *value* parameter, you set a local and temporary timeout: The new value is applied to the calling process only (the other processes keep the default values) and is reset to default as soon as the server receives any signal of activity from the client — for example, when the operation is finished. This option is useful for managing long operations initiated by 4D plug-ins.<br>To set the "No timeout" option, pass 0 in *value*. See example 1. |
| 4D Remote mode timeout | Longint | 14 | **Scope** (legacy network layer only): 4D application if *value* positive<br>**Kept between two sessions**: Yes if *value* positive<br>**Description**: To be used in very specific cases. Value of the timeout granted by the remote 4D machine to the 4D Server machine. The default timeout value used by 4D in remote mode is set on the "Client-Server/Network options" page of the Database settings dialog box on the remote machine.<br>The 4D Remote mode timeout selector is only taken into account if you are using the legacy network. It is ignored when the *ServerNet* layer is activated: this setting is entirely managed by the 4D Server timeout (13) selector. |
| Port ID | Longint | 15 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: No<br>**Description**: TCP port ID used by the 4D Web server with 4D in local mode and 4D Server. The default value, which can be set on the "Web/Configuration" page of the Preferences dialog box, is 80. You can use the constants of the **TCP Port Numbers** theme for the *value* parameter.<br>The Port ID selector is useful for 4D Web Servers compiled and merged with 4D Desktop (in which there is no access to the Design mode). For more information about the TCP port ID, refer to the **Web Server Settings** section. |
| IP Address to listen | Longint | 16 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Character set | Longint | 17 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Max concurrent Web processes | Longint | 18 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Client minimum Web process | Longint | 19 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 6<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client maximum Web process | Longint | 20 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 7<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client Max Web requests size | Longint | 21 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 27<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client port ID | Longint | 22 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 15<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client IP address to listen | Longint | 23 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 16<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client character set | Longint | 24 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 17<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client max concurrent Web proc | Longint | 25 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 18<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Maximum Web requests size | Longint | 27 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| 4D Server log recording | Longint | 28 | **Scope**: 4D Server, 4D remote<br>**Kept between two sessions**: No<br>**Possible values**: 0 or from 1 to X (0 = do not record, 1 to X = sequential number, added to the file name).<br>**Description**: Starts or stops the recording of standard requests received by 4D Server (excluding Web requests). By default, the value is 0 (requests not recorded).<br>4D Server lets you record each request received by the server machine in a log file. When this mechanism is enabled, two files are created in the Logs folder of the database, next to the database structure file.They are named 4DRequestsLog_X.txt and 4DRequestsLog_ProcessInfo_X.txt, where X is the sequential number of the log. Once the file 4DRequestsLog has reached a size of 10 MB, it is closed and a new one is generated, with an incremented sequential number. If a file of the same name already exists, it is replaced directly. You can set the starting number of the sequence using the *value* parameter.<br>These text files store various information concerning each request in a simple tabbed format: time, process number, size of request, processing duration, etc. This information is particularly useful during the development phase of the application or for statistical purposes. It can be imported, for example, into a spreadsheet software in order to be processed. |
| _o_Web Log recording | Longint | 29 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Client Web log recording | Longint | 30 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 = Do not record (default), 1 = Record in CLF format, 2 = Record in DLF format, 3 = Record in ELF format, 4 = Record in WLF format.<br>**Description**: Starts or stops the recording of Web requests received by the Web servers of all the client machines. By default, the value is 0 (requests not recorded).<br>The operation of this selector is identical to that of selector 29; however, it applies to all the 4D remote machines used as Web servers. The "logweb.txt" file is, in this case, automatically placed in the Logs subfolder of the remote 4D database folder (cache folder). If you only want to set values for certain client machines, use the Preferences dialog box of 4D in remote mode. |
| Table sequence number | Longint | 31 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Possible values**: Any longint value.<br>**Description**: This selector is used to modify or get the current unique number for records of the table passed as parameter. "Current number" means "last number used": if you modify this value using **SET DATABASE PARAMETER**, the next record will be created with a number that consists of the value passed + 1. This new number is the one returned by the **Sequence number** command as well in any field of the table to which the "Autoincrement" property has been assigned in the Structure editor or via SQL.<br>By default, this unique number is set by 4D and corresponds to the order of record creation. For additional information, refer to the documentation of the **Sequence number** command. |
| _o_Real display precision | Longint | 32 | **** *Selector disabled* **** |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Debug log recording | Longint | 34 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Description**: Starts or stops the sequential recording of events occurring at the 4D programming level in the *4DDebugLog* file, which is automatically placed in the Logs subfolder of the database, next to the structure file. A new, more compact, tabbed text format is used in the event log file "4DDebugLog[_n].txt" starting with 4D v14 (where _n is the segment number of the file).<br>**Possible values**: Longint containing a bit field: value = bit1(1)+bit2(2)+bit3(4)+bit4(8)+⋯).<br>- Bit 1 (value 1) requests to enable the file (note that any other non-null value also enables it as well)<br>- Bit 2 (value 2) requests call parameters to methods and commands.<br>- Bit 3 (value 4) enables new tabbed format.<br>- Bit 4 (value 8) disables immediate writing of each operation on disk (enabled by default). Immediate writing is slower but more effective, for example for investigating causes of a crash. If you disable this mode, the file contents are more compact and are generated more quickly.<br>- Bit 5 (value 16) disables recording of plug-in calls (enabled by default).<br>In the (former) non-tabbed format, execution times are expressed in milliseconds and the "< ms" value is displayed when an operation lasts less than one millisecond.<br>In the new tabbed format, execution times are expressed in microseconds.<br>Examples:<br>SET DATABASE PARAMETER (34;1) // enables mode v13 file without parameters, with runtimes<br>SET DATABASE PARAMETER (34;2) // enables mode v13 file with parameters and runtimes<br>SET DATABASE PARAMETER (34;2+4) // enables file with v14 format, with parameters and runtimes<br>SET DATABASE PARAMETER (34;0) // disables file<br>To avoid having a file record too much information, you can restrict the 4D commands that are examined by using selector 80, Log Command list.<br>This option can be enabled for any type of 4D application (4D all modes, 4D Server, 4D Volume Desktop), in interpreted or compiled mode.<br>**Note:** This option is provided solely for the purpose of debugging and must not be put into production since it may lead to deterioration of the application performance and saturation of the hard disk. For more information about this format and on the use of the 4DDebugLog[_n].txt file, please contact the Technical Support of 4D Inc. |
| Client Server port ID | Longint | 35 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 to 65535<br>**Description**: TCP port number where the 4D Server publishes the database (bound for 4D remote machines). By default, the value is 19813.<br>Customizing this value means that several 4D client-server applications can be used on the same machine with the TCP protocol; in this case, you must indicate a different port number for each application.<br>The value is stored in the database structure file. It can be set with 4D in local mode but is only taken into account in client-server configuration.<br>When you modify this value, it is necessary to restart the server machine in order for the new value to be taken into account. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Invert objects | Longint | 37 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0, 1 or 2 (0 = mode disabled, 1 = automatic mode, 2 = mode enabled).<br>**Description**: Configuration of the "object inversion" mode which is used to invert forms, objects, menu bars, and so on, in Application mode when the database is displayed under Windows in a right-to-left language. This mode can also be configured on the Interface/Right-to-left languages page of the Database Settings.<br>• Value 0 indicates that the mode is never enabled, regardless of the system configuration (corresponds to the Never value in the Database Settings).<br>• Value 1 indicates that the mode is enabled or disabled depending on the system configuration (corresponds to the Automatic value in the Database Settings).<br>• Value 2 indicates that the mode is enabled, regardless of the system configuration (corresponds to the Always value in the Database Settings).<br>For more information, refer to the *Design Reference* manual of 4D. |
| HTTPS Port ID | Longint | 39 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Client HTTPS port ID | Longint | 40 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 to 65535<br>**Description**: TCP port number used by the Web servers of the client machines for secure connections via SSL (HTTPS protocol). By default, the value is 443 (standard value).<br>This selector can be used to modify by programming the TCP port used by the Web servers of the client machines for secure connections via SSL (HTTPS protocol). By default, the value is 443 (standard value).<br>This selector operates exactly the same way as selector 39; however, it applies to all the 4D remote machines used as Web servers. If you only want to modify the value of certain specific client machines, use the Preferences dialog box of the remote 4D. |
| Unicode mode | Longint | 41 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (compatibility mode) or 1 (Unicode mode)<br>**Description**: Current database operating mode, with regards to the character set. 4D supports the Unicode character set but can function in “compatibility” mode (based on the Mac ASCII character set). By default, converted databases are executed in compatibility mode (0) and databases created with version 11 or higher are executed in Unicode mode. The execution mode can be controlled via an option in the Preferences and can also be read or (for testing purposes) modified via this selector. Modifying this option requires the database to be restarted in order for it to be taken into account. Note that within a component it is not possible to modify this value, but only to read it. |
| SQL Autocommit | Longint | 43 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (deactivation) or 1 (activation)<br>**Description**: Activation or deactivation of the SQL auto-commit mode. By default, the value is 0 (deactivated mode)<br>The auto-commit mode is used to strengthen the referential integrity of the database. When this mode is active, all **SELECT**, **INSERT**, **UPDATE** and **DELETE** (SIUD) queries are automatically included in ad hoc transactions when they are not already executed within a transaction. This mode can also be set in the Preferences of the database. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| SQL Engine case sensitivity | Longint | 44 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (case not taken into account) or 1 (case-sensitive)<br>**Description**: Activation or deactivation of case-sensitivity for string comparisons carried out by the SQL engine.<br>By default, the value is 1 (case-sensitive): the SQL engine differentiates between upper and lower case and between accented characters when comparing strings (sorts and queries). For example “ABC”= “ABC” but “ABC” # “Abc” and "abc" # "âbc." In certain cases, for example so as to align the functioning of the SQL engine with that of the 4D engine, you may wish for string comparisons to not be case-sensitive (“ABC”=“Abc"="âbc").<br>This option can also be set on the **SQL page** of the Database settings. |
| Client log recording | Longint | 45 | **Scope**: Remote 4D machine<br>**Kept between two sessions**: No<br>**Possible values**: 0 or from 1 to X (0 = do not record, 1 to X = sequential number, attached to file name).<br>**Description**: Starts or stops recording of standard requests carried out by the 4D client machine that executed the command (excluding Web requests). By default, the value is 0 (no recording of requests).<br>4D lets you record the log of requests carried out by the client machine. When this mechanism is activated, two files are created on the client machine, in the Logs subfolder of the local folder of the database. They are named 4DRequestsLog_X.txt and 4DRequestsLog_ProcessInfo_X.txt, where X is the sequential number of the log. Once the file 4DRequestsLog has reached a size of 10 MB, it is closed and a new one is generated, with an incremented sequential number. If a file with the same name already exists, it is directly replaced. You can set the starting number for the sequence using the value parameter.<br>These text files store various information concerning each request in a simple tabbed format: time, process number, size of request, processing duration, etc. This information is particularly useful during the development phase of the application or for statistical purposes |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Query by formula on server | Longint | 46 | **Scope**: Current table and process<br>**Kept between two sessions**: No<br>**Possible values**: 0 (use database configuration), 1 (execute on client) or 2 (execute on server)<br>**Description**: Execution location of **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** commands for the *table* passed in the parameter.<br>When using a database in client-server mode, the query "by formula" commands can be executed either on the server or on the client machine:<br>• In databases created with 4D v11 SQL, these commands are executed on the server.<br>• In converted databases, these commands are executed on the client machine, as in previous versions of 4D.<br>• In converted databases, a specific preference (Application/Compatibility page) can be used to globally modify the execution location of these commands.<br>This difference in execution location influences not only application performance (execution on the server is usually faster) but also programming. In fact, the value of the components of the formula (in particular variables called via a method) differ according to the execution context. You can use this selector to punctually adapt the operation of your application.<br>If you pass 0 in the *value* l'parameter, the execution location of query "by formula" commands will depend on the database configuration: in databases created with 4D v11 SQL, these commands will be executed on the server. In converted databases, they will be executed on the client machine or the server according to the database preferences. Pass 1 or 2 in *value* to "force" the execution of these commands, respectively, on the client or on the server machine.<br>Refer to example 4.<br>**Note:** If you want to be able to enable "SQL type" joins (see the QUERY BY FORMULA Joins selector), you must always execute formulas on the server so that they have access to the records. Be careful, in this context, the formula must not contain any calls to a method, otherwise it will automatically be switched to the remote machine. |
| Order by formula on server | Longint | 47 | **Scope**: Current table and process<br>**Kept between two sessions**: No<br>**Possible values**: 0 (use database configuration), 1 (execute on client) or 2 (execute on server)<br>**Description** : Execution location of **ORDER BY FORMULA** command for the table passed in the parameter.<br>When using a database in client-server mode, this command can be executed either on the server or on the client machine. This selector can be used to specify the execution location of this command (server or client). This mode can also be set in the database preferences. For more information, please refer to the description of selector 46, Query By Formula On Server.<br>**Note:** If you want to be able to enable "SQL type" joins (see the QUERY BY FORMULA Joins selector), you must always execute formulas on the server so that they have access to the records. Be careful, in this context, the formula must not contain any calls to a method, otherwise it will automatically be switched to the remote machine. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Auto synchro resources folder | Longint | 48 | **Scope**: 4D remote machine<br>**Kept between two sessions**: No<br>**Possible values**: 0 (no synchronization), 1 (auto synchronization) or 2 (ask).<br>**Description**: Dynamic synchronization mode for *Resources* folder of 4D client machine that executed the command with that of the server.<br>When the contents of the *Resources* folder on the server has been modified or a user has requested synchronization (for example via the resources explorer or following the execution of the **SET DATABASE LOCALIZATION** command), the server notifies the connected client machines.<br>Three synchronization modes are then possible on the client side. The Auto Synchro Resources Folder selector is used to specify the mode to be used by the client machine for the current session:<br>• 0 (default value): no dynamic synchronization (synchronization request is ignored)<br>• 1: automatic dynamic synchronization<br>• 2: display of a dialog box on the client machines, with the possibility of allowing or refusing synchronization.<br>The synchronization mode can also be set globally in the application Preferences. |
| Query by formula joins | Longint | 49 | **Scope**: Current process<br>**Kept between two sessions**: No<br>**Possible values**: 0 (use database configuration), 1 (always use automatic relations) or 2 (use SQL joins if possible).<br>**Description**: Operating mode of the **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** commands relating to the use of "SQL joins."<br>In databases created starting with version 11.2 of 4D v11 SQL, these commands carry out joins based on the SQL joins model. This mechanism can be used to modify the selection of a table according to a query carried out on another table without these tables being connected by an automatic relation (necessary condition in previous versions of 4D).<br>The QUERY BY FORMULA Joins selector lets you specify the operating mode of the query by formula commands for the current process:<br>• 0: Uses the current settings of the database (default value). In databases created starting with version 11.2 of 4D v11 SQL, "SQL joins" are always activated for queries by formula. In converted databases, this mechanism is not activated by default for compatibility reasons but can be implemented via a preference.<br>• 1: Always use automatic relations (= functioning of previous versions of 4D). In this mode, a relation is necessary in order to set the selection of a table according to queries carried out on another table. 4D does not do "SQL joins."<br>• 2: Use SQL joins if possible (= default operation of databases created in version 11.2 and higher of 4D v11 SQL ). In this mode, 4D establishes "SQL joins" for queries by formula when the formula is suited for it (with two notable exceptions, see the description of the **QUERY BY FORMULA** or **QUERY SELECTION BY FORMULA** command).<br>**Note:** With 4D in remote mode, "SQL joins" can only be used if the formulas are executed on the server (they must have access to the records). To configure where formulas are to be executed, please refer to selectors 46 and 47. |
| HTTP compression level | Longint | 50 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| HTTP compression threshold | Longint | 51 | **Scope**:4D application<br>**Kept between two sessions**: No<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Server base process stack size | Longint | 53 | **Scope**: 4D Server<br>**Kept between two sessions**: No<br>**Possible values**: Positive longint.<br>**Description**: Size of the stack allocated to each preemptive system process on the server, expressed in bytes. The default size is determined by the system.<br>Preemptive system processes (processes of the 4D client base process type) are loaded to control the main 4D client processes. The size allocated by default to the stack of each preemptive process allows a good ease of execution but may prove to be consequential when very large numbers of processes (several hundred) are created. For optimization purposes, this size can be reduced considerably if the operations carried out by the database allow for it (for example if the database does not carry out sorts of large quantities of records). Values of 512 or even 256 KB are possible. Be careful, under-sizing the stack is critical and can be harmful to the operation of 4D Server. Setting this parameter should be done with caution and must take the database conditions of use into account (number of records, type of operations, etc.).<br>In order to be taken into account, this parameter must be executed on the server machine (for example in the **On Server Startup Database Method**). |
| Idle connections timeout | Longint | 54 | **Scope**:4D application unless value is negative<br>**Kept between two sessions**: No<br>**Possible values**: Whole value expressing a duration in seconds. The value can be positive (new connections) or negative (existing connections). By default, the value is 20.<br>**Description**: Maximum period of inactivity (timeout) for connections to both the 4D database engine and the SQL engine, as well as, in *ServerNet* mode (new network layer), to the 4D application server. When an idle connection reaches this limit, it is automatically put on standby, which freezes the client/server session and closes the network socket. In the server administration window, the state of the user process is indicated as "Postponed". This functioning is completely transparent for the user: as soon as there is new activity on the connection which is on standby, the socket is automatically reopened and the client/server session is restored.<br>On the one hand, this setting lets you save resources on the server: connections on standby close the socket and free up a process on the server. On the other hand, it lets you avoid losing connections due to the closing of idle sockets by the firewall. For this, the timeout value for idle connections must be lower than that of the firewall in this case.<br>If you pass a positive value in *value*, it applies to all new connections in all the processes. If you pass a negative value, it applies to connections that are open in the current process. If you pass 0, idle connections are not subjected to a timeout.<br>This parameter can be set on both the server and client side. If you pass two different durations, the shorter one is taken into account. Usually, you do not need to change this value. |
| PHP interpreter IP address | Longint | 55 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Formatted string of the type "nnn.nnn.nnn.nnn" (for example "127.0.0.1").<br>**Description**: IP address used locally by 4D to communicate with the PHP interpreter via FastCGI. By default, the value is "127.0.0.1". This address must correspond to the machine where 4D is located. This parameter can also be set globally for all the machines via the Database Settings.<br>For more information about the PHP interpreter, please refer to the *Design Reference* manual. |
| PHP interpreter port | Longint | 56 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Positive long integer type value. By default, the value is 8002.<br>**Description**: Number of the TCP port used by the PHP interpreter of 4D. This parameter can also be modified globally for all the machines via the Database Settings.<br>For more information about the PHP interpreter, please refer to the *Design Reference* manual. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| PHP number of children | Longint | 57 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Positive long integer type value. By default, the value is 5.<br>**Description**: Number of child processes to be created and maintained locally by the PHP interpreter of 4D. For optimization reasons, the PHP interpreter creates and uses a set (pool) of system processes called "child processes" for processing script execution requests. You can vary the number of child processes according to the needs of your application. This parameter can also be modified globally for all the machines via the Database Settings. For more information about the PHP interpreter, please refer to the *Design Reference* manual.<br>**Note:** Under Mac OS, all the child processes share the same port. Under Windows, each child process uses a specific port number. The first number is the one set for the PHP interpreter; the other child processes increment the first one. For example, if the default port is 8002 and you launch 5 child processes, they will use ports 8002 to 8006. |
| PHP max requests | Longint | 58 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Positive long integer type value. By default, the value is 500.<br>**Description**: Maximum number of requests accepted by the PHP interpreter. When this maximum number is reached, the interpreter returns errors of the "server busy" type. For security or performance reasons, you can modify this value. This parameter can also be modified globally for all the machines via the Database Settings. For more information about this parameter, please refer to the FastCGI-PHP documentation.<br>**Note:** On the 4D side, these parameters are applied dynamically; it is not necessary to exit 4D in order for them to be taken into account. On the other hand, if the PHP interpreter is already launched, it will be necessary to exit and relaunch it in order for these modifications to be taken into account. |
| PHP use external interpreter | Longint | 60 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values** : 0 = use internal interpreter, 1 = use external interpreter<br>**Description**: Value indicating whether PHP requests in 4D are sent to the internal interpreter provided by 4D or to an external interpreter. By default the value is 0 (use of interpreter provided by 4D). If you want to use your own PHP interpreter, for example in order to use additional modules or a specific configuration, pass 1 in *value*. In this case, 4D does not launch its internal interpreter in the case of PHP requests.<br>The custom PHP interpreter must have been compiled in FastCGI and be located on the same machine as the 4D engine. Note that in this case, you must manage the interpreter entirely; it will not be started nor stopped by 4D. This parameter can also be modified globally for all the machines via the Database Settings. |
| Maximum temporary memory size | Longint | 61 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Positive longint.<br>**Description**: Maximum size of temporary memory that 4D can allocate to each process, expressed in MB. By default, the value is 0 (no maximum size). 4D uses a special temporary memory dedicated to indexing and sorting operations. This memory is intended to preserve the "standard" cache memory during massive operations. It is activated only when needed. By default, the size of the temporary memory is limited only by the resources available (according to the system memory configuration).<br>This mechanism is suitable for most applications. However, in certain specific contexts, more particularly when a client-server application simultaneously carries out a large number of sequential sorts, the size of the temporary memory can increase critically, to the point where it can render the system unstable. In this context, setting a maximum size for the temporary memory allows you to preserve proper functioning of the application. In return, the running speed might be affected: when the maximum size is reached for a process, 4D uses disk files which may slow down processing. For specific needs such as those described above, a maximum size of around 50 MB is generally a good compromise. However, the ideal value will need to be determined according to the specificities of the application and will generally be the result of real-time volumetric testing. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| SSL cipher list | String | 64 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Sequence of strings separated by colons (for example "RC4-MD5:RC4-64-MD5:....")<br>**Description:** Cipher list used by 4D for the secure protocol. This list modifies the priority of ciphering algorithms implemented by 4D. For example, you can pass the following string in the *value* parameter: "AES:ALL:!aNULL:!eNULL:+RC4:@STRENGTH". For a complete description of the syntax for the ciphers list, refer to the ciphers page of the OpenSSL site.<br>This setting applies to the entire 4D application (it concerns the HTTP server, SQL server, client/server connections, as well as the HTTP client and all the 4D commands that make use of the secure protocol) but it is temporary (it is not maintained between sessions).<br>When the cipher list has been modified, you will need to restart the server concerned in order for the new settings to be taken into account.<br>To reset the cipher list to its default value (stored permanently in the SLI file), call the **SET DATABASE PARAMETER** command and pass an empty string ("") in the *value* parameter.<br>**Note:** With the **Get database parameter** command, the cipher list is returned in the optional *stringValue* parameter and the return parameter is always 0. |
| Cache unload minimum size | Longint | 66 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Positive longint > 1.<br>**Description**: Minimum size of memory to release from the database cache when the engine needs to make space in order to allocate an object to it (value in bytes).<br>The purpose of this selector is to reduce the number of times that data is released from the cache in order to obtain better performance. You can vary this setting according to the size of the cache and that of the blocks of data being handled in your database.<br>By default, if this selector is not used, 4D unloads at least 10% of the cache when space is needed. |
| Direct2D status | Longint | 69 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Description**: Activation mode to implement Direct2D under Windows.<br>**Possible values**: One of the following constants (mode 3 by default):<br>Direct2D Disabled (0): Direct2D mode is not enabled and the database functions in the previous mode (GDI/GDIPlus).<br>Direct2D Hardware (1): Use Direct2D as graphics hardware context for entire 4D application. If this context is not available, use Direct2D graphics software context (except under Vista, in which case GDI/GDIPlus mode is used for better performance).<br>Direct2D Software (3) (Default mode): Beginning with Windows 7, use Direct2D graphics software context for entire 4D application. Under Vista, GDI/GDIPlus mode is used for better performance.<br>**Compatibility note:** Starting with 4D v14, hybrid modes are disabled and redirected to available modes (the former mode 2 is equivalent to 1; former modes 4 and 5 are equivalent to mode 3). |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Direct2D get active status | Longint | 74 | **Note**: You can only use this selector with the **Get database parameter** command and its value cannot be set.<br>**Description**: Returns active implementation of Direct2D under Windows.<br>**Possible values**: 0, 1, 2, 3, 4 or 5 (see values of selector 69). The value returned depends on the availability of Direct2D, the hardware and the quality of Direct2D support by the operating system.<br>For example, if you execute:<br>`SET DATABASE PARAMETER(Direct2D status;Direct2D Hardware)`<br>`$mode:=Get database parameter(Direct2D get active status)`<br><br>- On Windows 7 and higher, *$mode* is set to 1 when the system detects hardware compatible with Direct2D; otherwise, *$mode* is set to 3 (software context).<br>- On Windows Vista, *$mode* is set to 1 when the system detects hardware compatible with Direct2D; otherwise, *$mode* is set to 0 (disabling of Direct2D).<br>- On Windows XP, *$mode* is always set to 0 (not compatible with Direct2D). |
| Diagnostic log recording | Longint | 79 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: 0 or 1 (0 = do not record, 1 = record)<br>**Description**: Starts or stops recording of the 4D diagnostic file. By default, the value is 0 (do not record).<br>4D can continuously record a set of events related to the internal application operation into a diagnostic file. Information contained in this file is intended for the development of 4D applications and can be analyzed with the help of the 4D tech support. When you pass 1 in this selector, a diagnostic file, named *DatabaseName_X*.txt, is automatically created (or opened) in the database **Logs** folder. Once this file reaches a size of 10 MB, it is closed and a new file named *DatabaseName_X*.txt is generated, with an incremented sequence number X.<br>Note that you can include custom information in this file using the **LOG EVENT** command. |
| Log command list | String | 80 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: String containing a list of 4D command numbers to record (separated by semi-colons) or "all" to record all the commands or "" (empty string) to record none of them.<br>**Description**: List of 4D commands to record in the debugging file (see selector 34, Debug Log Recording). By default, all 4D commands are recorded.<br>This selector restricts the quantity of information saved in the debugging file by limiting the 4D commands whose execution you want to record. For example, you can write:<br>`SET DATABASE PARAMETER(Log command list;"277;341") //Record only the QUERY and QUERY SELECTION commands` |
| Spellchecker | Longint | 81 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: 0 (default) = native OS X spellchecker (Hunspell disabled), 1 = Hunspell spellcheck enabled.<br>**Description**: Enables the Hunspell spellcheck under OS X. By default, the native spellchecker is enabled on this platform. You may prefer to use the Hunspell spellcheck, for example, in order to unify the interface for your cross-platform applications (under Windows, only the Hunspell spellcheck is available). For more information, refer to **Support of Hunspell dictionaries**. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| QuickTime support | Longint | 82 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (default) = QuickTime disabled, 1 = QuickTime enabled.<br>**Description**: In 4D starting with v14, by default QuickTime codecs are no longer supported. For compatibility, you can use this selector to re-enable them in your database. Modification of this option requires that the database be restarted. Nevertheless, you should note that in future versions of 4D, QuickTime support is permanently removed. |
| JSON use local time | Longint | 85 | **Scope**: Current process<br>**Kept between two sessions**: No<br>**Possible values**: 0 = ignore local time zone, 1 (default) = take time zone into account.<br>**Description**: By default, 4D dates converted to JSON format take the local time zone into account. For example, converting the date !23/08/2013! gives you "2013-08-22T22:00:00Z" in JSON format when the operation is performed in France during Daylight Savings Time (GMT+2). This principle conforms to the standard operation of JavaScript.<br>This can be a source of errors when you want to send JSON date values to someone in a different time zone. This is the case, for example, when you export a table using **Selection to JSON** in France that is meant to be reimported in the US using **JSON TO SELECTION**. By default, since dates are re-interpreted in each time zone, the values stored in the database will be different. In this case, you can modify the conversion mode for dates so that they do not take the time zone into account by passing 0 in this selector. Converting the date !23/08/2013! will then give you "2013-08-23T00:00:00Z" in all cases. |
| Use legacy network layer | Longint | 87 | **Scope**: 4D in local mode, 4D Server<br>**Kept between two sessions**: Yes<br>**Description:** Sets or gets the current status of the legacy network layer for client/server connections. The legacy network layer is obsolete beginning with 4D v14 R5 and should be replaced progressively in your applications with the *ServerNet* network layer. *ServerNet* will be required in upcoming 4D releases in order to benefit from future network evolutions. For compatibility reasons, the legacy network layer is still supported to allow a smooth transition for existing applications; (it is used by default in applications converted from a release prior to v14 R5). Pass 1 in this parameter to use the legacy network layer (and disable *ServerNet*) for your client/server connections, and pass 0 to disable the legacy network (and use the *ServerNet*).<br>This property can also be set by means of the "Use legacy network layer" option found on the **Compatibility page** of the Database Settings (see **Network and Client-Server options**). In this section, you will also find a discussion about migration strategy. We recommend that you activate the *ServerNet* as soon as possible.<br>You will need to restart the application in order for this parameter to be taken into account. It is not available in 4D Server v14 R5 64-bit version for OS X, which only supports the *ServetNet*; (it always returns 0).<br>**Possible values:** 0 or 1 (0 = do not use legacy layer, 1 = use legacy layer)<br>**Default value:** 0 in databases created with 4D v14 R5 or higher, 1 in databases converted from 4D v14 R4 or earlier. |
| SQL Server Port ID | Longint | 88 | **Scope**: 4D local, 4D Server.<br>**Kept between two sessions**: Yes<br>**Description**: Gets or sets the TCP port number used by the integrated SQL server of 4D in local mode or 4D Server. By default, the value is 19812. When this selector is set, the database setting is updated. You can also set the TCP port number on the "SQL" page of the Database Settings dialog box.<br>**Possible values**: 0 to 65535.<br>**Default value**: 19812 |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Circular log limitation | Longint | 90 | **Scope**: 4D local, 4D Server.<br>**Kept between two sessions**: No<br>**Possible values**: Any integer value, 0 = keep all logs<br>**Description**: Maximum number of files to keep in rotation for each type of log. By default, all files are kept. If you pass a value *X*, only the *X* most recent files are kept, with the oldest being erased automatically when a new one is created. This setting applies to each of the following log files: request logs (selectors 28 and 45), debug log (selector 34), events log (selector 79), as well as Web request logs and Web debug logs (selectors 29 and 84 of the **WEB SET OPTION** command). |
| Number of formulas in cache | Longint | 92 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Positive longints<br>**Default value**: 0 (no cache)<br>**Description**: Sets or gets the maximum number of formulas to be kept in the cache of formulas, which is used by the **EXECUTE FORMULA** command. This limit is applied to all processes, but each process has its own formula cache. Caching formulas accelerates the **EXECUTE FORMULA** command execution in compiled mode since each cached formula is tokenized only once in this case. When you change the cache value, existing contents are reset even if the new size is larger than the previous one. Once the maximum number of formulas in the cache is reached, a new executed formula will erase the oldest one in the cache (FIFO mode). This parameter is only taken into account in compiled databases or compiled components. |
| Cache flush periodicity | Longint | 95 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: No<br>**Possible values**: longint > 1 (seconds)<br>**Description**: Gets or sets the current cache flush periodicity, expressed in seconds. Modifying this value overrides the **Flush Cache every X Seconds** option in the **Database/Memory page** of the Database settings for the session (it is not stored in the Database settings). |

**Note**: The *table* parameter is only used by selectors 31, 32, 46 and 47. In all other cases, it is ignored if it is passed.

## Example 1

The following statement will avoid any unexpected timeout:

```
 `Increasing the timeout to 3 hours for the current process
SET DATABASE PARAMETER(4D Server Timeout;-60*3)
 `Executing a time-consuming operation with no control from 4D
...
WR PRINT MERGE(Area;3;0)
...
```

## Example 2

This example temporarily forces the execution of a query by formula command on the client machine:

```
curVal:=Get database parameter([table1];Query By Formula On Server) `Store the current setting
SET DATABASE PARAMETER([table1];Query By Formula On Server;1) `Force execution on the client machine
QUERY BY FORMULA([table1];myformula)
SET DATABASE PARAMETER([table1];Query By Formula On Server;curVal) `Re-establish current setting
```

## Example 3

You want to export data in JSON that contains a converted 4D date. Note that conversion occurs when the date is saved in the object, so you must call the **SET DATABASE PARAMETER** command before calling **OB SET**:

```
C_OBJECT($o)
SET DATABASE PARAMETER(JSON use local time;0)
```

```
OB SET($o ;"myDate";Current date) // JSON conversion
$json:=JSON Stringify($o)
SET DATABASE PARAMETER(JSON use local time;1)
```

# SET UPDATE FOLDER

SET UPDATE FOLDER ( folderPath {; silentErrors} )

| Parameter | Type | | Description |
|---|---|---|---|
| folderPath | String | ⇒ | Pathname of folder (package under OS X) containing updated application |
| silentErrors | Boolean | ⇒ | False (default) = report errors visibly, True = do not report them |

## Description

The **SET UPDATE FOLDER** command specifies the folder containing the update of the current merged 4D application. This information is stored in the 4D session until the **RESTART 4D** method is called. If the application is exited manually, this information is not kept.

This command is intended to be used in an automatic update process for a merged application (server or single-user). For more information, refer to the **Finalizing and deploying final applications** section in the *Design Reference* manual.

**Note:** This command only works with 4D Server or a single-user application merged with 4D Volume Desktop.

In the *folderPath* parameter, pass the complete pathname for the folder of the new version of the merged application (folder containing the *my4DApp.exe* application under Windows or the *my4DApp.app* package under OS X), created by the 4D application builder.

**Note:** We recommend that you use the same names for the files in the new version of the application as the ones in the original, since the application folder is replaced during the update. If you use different names for these files, any stored shortcuts and/or paths will no longer work.

If the parameters are valid, the update is placed "on hold" in the session until the **RESTART 4D** command is called. If you executed **SET UPDATE FOLDER** several times before calling **RESTART 4D**, the last valid call is taken into account.

In case of anomaly, an error is generated; the *silentErrors* parameter determines whether or not these errors are displayed (see below).

You can pass an empty string ("") in the *folderPath* parameter to reset the update information for the current session.

The optional *silentErrors* parameter specifies how errors are reported during the update:

- When you pass **False** or when this parameter is omitted, errors are recorded in the update journal and displayed in a warning dialog box.
- If you pass **True**, errors are simply recorded in the update journal.

Exception: if the journal file cannot be created, a warning dialog box is displayed, regardless of the value of the *silentErrors* parameter. For more information, refer to the description of the **Get last update log path** command.

If the command is executed correctly, the OK system variable is set to 1; otherwise, it is set to 0. You can intercept any errors generated by the command using a method installed using the **ON ERR CALL** command.

## Example

You created a "MyUpdates" folder on your disk, where you placed a new version of the "MyApp" application. You do not want to display errors. To prepare the update, you write:

```
 // Windows syntax
SET UPDATE FOLDER("C:¥¥MyUpdates"+Folder separator+"MyApp"+Folder separator;True)

 // OS X syntax
SET UPDATE FOLDER("MacHD:MyUpdates"+Folder separator+"MyApp.app"+Folder separator;True)
```

# Structure file

Structure file {( * )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | Returns structure file of host database |
| Function result | String | ⮌ | Long name of the database structure file |

## Description

The **Structure file** command returns the long name of the structure file for the database with which you are currently working.

*On Windows*
If, for example, you are working with the database MyCDs located in ¥DOCS¥MyCDs on the volume G, the command returns G:¥DOCS¥MyCDs¥MyCDs.4DB.

*On Macintosh*
If, for example, you are are working with the database located in the folder Documents:MyCDsƒ: on the disk Macintosh HD, the command returns Macintosh HD:Documents:MyCDsƒ:MyCDs.

**Note:** In the particular case of a database that has been compiled and merged with 4D Volume Desktop, this command returns the pathname of the application file (executable application) under Windows and OS X. Under OS X, this file is located inside the software package, in the [Contents:Mac OS] folder. This stems from a former mechanism and is kept for compatibility reasons. If you want to get the full name of the software package itself, it is preferable to use the **Application file** command. The technique consists of testing the application using the **Application type** command, then executing **Structure file** or **Application file** depending on the context.

**WARNING:** If you call this command when using 4D in remote mode, only the name of the structure file is returned; the long name is not returned.

The optional * parameter is useful in the case of an architecture using components: it can be used to determine the structure (host or component) for which you want to get the long name depending on the context in which the command is called:

- When the command is called from a component:
  - If the * parameter is passed, the command returns the long name of the structure file of the host database,
  - If the * parameter is not passed, the command returns the long name of the structure file of the component. The structure file of the component corresponds to the .4db or .4dc file of the component found in the “Components” folder of the database. However, a component can also be installed as an alias/shortcut or a .4dbase folder/package:
    - In the case of a component installed as an alias/shortcut, the command returns the pathname of the original .4db or .4dc file (the alias or shortcut is resolved).
    - In the case of a component installed as a .4dbase folder/package, the command returns the pathname of the .4db or .4dc file located within this folder/package.
- When the command is called from a method of the host database, it always returns the long name of the structure file of the host database, regardless of whether or not the * parameter is passed.

## Example 1

This example displays the name and the location of the structure file currently in use:

```
If(Application type#4D Remote mode)
   $vsStructureFilename:=Long name to file name(Structure file)
   $vsStructurePathname:=Long name to path name(Structure file)
   ALERT("You are currently using the database "+Char(34)+$vsStructureFilename+Char(34)+
   " located at "+Char(34)+$vsStructurePathname+Char(34)+".")
Else
   ALERT("You are connected to the database "+Char(34)+Structure file+Char(34))
End if
```

**Note:** The project methods **Long name to file name** and **Long name to path name** are listed in the section **System Documents**.

## Example 2

The following example can be used to find out whether the method is called from a component:

```
C_BOOLEAN($0)
$0:=(Structure file#Structure file(*))
  ` $0=True if method is called from a component
```

# VERIFY CURRENT DATA FILE

VERIFY CURRENT DATA FILE {( objects ; options ; method {; tablesArray {; fieldsArray}} )}

| Parameter | Type | | Description |
|---|---|---|---|
| objects | Longint | ⇒ | Objects to check |
| options | Longint | ⇒ | Checking options |
| method | Text | ⇒ | Name of 4D callback method |
| tablesArray | Longint array | ⇒ | Numbers of tables to be checked |
| fieldsArray | 2D Integer array, 2D Longint array, 2D Real array | ⇒ | Numbers of indexes to be checked |

## Description

The **VERIFY CURRENT DATA FILE** command carries out a structural check of the objects found in the data file currently opened by 4D.

This command has the same functioning as the **VERIFY DATA FILE** command, except that it only applies to the current data file of the open database. It therefore does not require parameters specifying the structure and data.

Refer to the **VERIFY DATA FILE** command for a description of the parameters.

If you pass the **VERIFY CURRENT DATA FILE** command with no parameters, the verification is carried out with the default values of the parameters:

- *objects* = Verify All (= value 16)
- *options* = 0 (log file is created but not timestamped)
- *method* = ""
- *tablesArray* and *fieldsArray* are omitted.

When this command is executed, the data cache is flushed and all operations accessing the data are blocked during the verification.

If a log file has been generated, its complete pathname is returned in the *Document* system variable.

## System variables and sets

If the callback method does not exist, the verification is not carried out, an error is generated and the system variable OK is set to 0. If a log file was generated, its complete pathname is returned in the Document system variable.

# VERIFY DATA FILE

VERIFY DATA FILE ( structurePath ; dataPath ; objects ; options ; method {; tablesArray {; fieldsArray}} )

| Parameter | Type | | Description |
|---|---|---|---|
| structurePath | Text | ➡ | Pathname of 4D structure file to be checked |
| dataPath | Text | ➡ | Pathname of 4D data file to be checked |
| objects | Longint | ➡ | Objects to be checked |
| options | Longint | ➡ | Checking options |
| method | Text | ➡ | Name of 4D callback method |
| tablesArray | Longint array | ➡ | Numbers of tables to be checked |
| fieldsArray | 2D Integer array, 2D Longint array, 2D Real array | ➡ | Numbers of indexes to be checked |

## Description

The **VERIFY DATA FILE** command carries out a structural check of the objects contained in the 4D data file designated by *structurePath* and *dataPath*.

**Note:** For more information about checking data, please refer to the Design Reference manual. *structurePath* designates the structure file (compiled or not) associated with the data file to be checked. This can be the open structure file or any other structure file. You must pass a complete pathname, expressed with the syntax of the operating system. You can also pass an empty string, in this case a standard Open file dialog box appears so that the user can specify the structure file to be used.

*dataPath* designates a 4D data file (.4DD). It must correspond to the structure file defined by the *structurePath* parameter. Be careful, you can designate the current structure file but the data file must not be the current (open) file. To verify the currently open data file, use the **VERIFY CURRENT DATA FILE** command. If you attempt to verify the current data file with the **VERIFY DATA FILE** command, an error is generated.

The data file designated is opened in read only. You must make sure that no application accesses this file in write mode, otherwise the results of the check may be distorted.

In the *dataPath* parameter, you can pass an empty string, a file name or a complete pathname, expressed in the syntax of the operating system. If you pass an empty string, the standard Open file dialog box appears so that the user can specify the file to be checked (note that in this case, it is not possible to select the current data file). If you only pass a data file name, 4D will look for it at the same level as the specified structure file.

The *objects* parameter is used to designate which types of objects will be checked. Two types of objects can be checked: records and indexes. You can use the following constants, found in the "**Data File Maintenance**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Verify all | Longint | 16 | |
| Verify indexes | Longint | 8 | This option checks the physical consistency of the indexes, without any link with the data. It signals invalid keys but does not permit you to detect duplicated keys (two indexes that point to the same record). This type of error can only be detected with the Verify All option. |
| Verify records | Longint | 4 | |

To verify both the records and the indexes, pass the total of Verify Records+Verify Indexes. The value 0 (zero) can also be used to obtain the same result. The Verify All option carries out complete internal verification. This verification is compatible with the creation of a log.

The *options* parameter is used to set verification options. The following options are available, found in the"**Data File Maintenance**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Do not create log file | Longint | 16384 | Generally, this command creates a log file in XML format (refer to the end of the command description). With this option, no log file will be created. |
| Timestamp log file name | Longint | 262144 | When this option is passed, the name of the log file generated will contain the date and time of its creation; as a result, it will not replace any log file already generated previously. By default, if this option is not passed, log file names are not timestamped and each new file generated replaces the previous one. |

Generally, the **VERIFY DATA FILE** command creates a log file in XML format (please refer to the end of the description of this command). You can cancel this operation by passing this option. To create the log file, pass 0 in *options*.

The *method* parameter is used to set a callback method that will be called regularly during the verification. If you pass an empty string, no method is called. If the method passed does not exist, the verification is not carried out, an error is generated and the OK variable is set to 0. When it is called, this method receives up to 5 parameters depending on the objects being verified and on the event type originating the call (see calls table). It is imperative to declare these parameters in the method:

| | | |
|---|---|---|
| - $1 | Longint | Message type (see table) |
| - $2 | Longint | Object type |
| - $3 | Text | Message |
| - $4 | Longint | Table number |
| - $5 | Longint | Reserved |

The following table describes the contents of the parameters depending on the event type:

| Event | $1 (Longint) | $2 (Longint) | $3 (Text) | $4 (Longint) | $5 (Longint) |
|---|---|---|---|---|---|
| Message | 1 | 0 | Progression message | Percentage done (0-100) | Reserved |
| Verification finished(*) | 2 | Object type (**) | OK message test | Table or index number | Reserved |
| Error | 3 | Object type (**) | Text of error-message | Table or index number | Reserved |
| End of execution | 4 | 0 | DONE | 0 | Reserved |
| Warning | 5 | Object type(**) | Text of error message | Table or index number | Reserved |

(*) The *Verification finished* ($1=2) event is never returned when the mode is Verify All. It is only used in Verify Records or Verify Indexes mode.
(**) *Object type*: When an object is verified, a "finished" message ($1=2), error ($1=3) or warning ($1=5) can be sent. The object type returned in $2 can be one of the following:

- 0 = undetermined
- 4 = record
- 8 = index
- 16 = structure object (preliminary check of data file).

*Special case*: When $4 = 0 for $1=2, 3 or 5, the message does not concern a table or an index but rather the data file as a whole.

The callback method must also return a value in $0 (Longint), which is used to check the execution of the operation:

- If $0 = 0, the operation continues normally
- If $0 = -128, the operation is stopped without any error generated
- If $0 = another value, the operation is stopped and the value passed in $0 is returned as the error number. This error can be intercepted by an error-handling method.

**Note:** You cannot interrupt execution via $0 after the *End of execution* event ($4=1) has been generated.

Two optional arrays can also be used by this command:

- The *tablesArray* array contains the numbers of the tables whose records are to be checked. It can be used to limit checking to only certain tables. If this parameter is not passed or if the array is empty and the *objects* parameter contains Verify Records, all the tables will be checked.
- The *fieldsArray* array contains the numbers of the indexed fields whose indexes are to be checked. If this parameter is not passed or if the array is empty and the *objects* parameter contains Verify Indexes, all the indexes will be checked. The command ignores fields that are not indexed. If a field contains several indexes, they are all checked. If a field is part of a composite index, the entire index is checked.
  You must pass a 2D array in *fieldsArray*. For each row of the array:
  - The element {0} contains the table number,
  - The other elements {1...x} contain the field numbers.

By default, the **VERIFY DATA FILE** command creates a log file in XML format (if you have not passed the Do not create log file option, see the *options* parameter). This file is placed in the **Logs** folder of the current database and its name is also based on the structure file of the current database. For example, for a structure file named “myDB.4db,” the log file will be named “myDB_Verify_Log.xml.”
If you have passed the Timestamp log file name option, the name of the log file includes the date and time of its creation in the form "YYYY-MM-DD HH-MM-SS", which gives us, for example: “myDB_Verify_Log_2015-09-27 15-20-35.xml”. This means that each new log file does not replace the previous one, but it might require subsequent manual action to remove unnecessary files.
Regardless of the option selected, as soon as a log file is generated, its path is returned in the *Document* system variable after execution of the command.

## Example 1

Simple checking of data and indexes:

```
VERIFY DATA FILE($StructName;$DataName;Verify indexes+Verify records;Do not create log file;"")
```

## Example 2

Complete verification with log file:

```
VERIFY DATA FILE($StructName;$DataName;Verify all;0;"")
```

## Example 3

Checking of records only:

```
VERIFY DATA FILE($StructName;$DataName;Verify records;0;"")
```

## Example 4

Checking of records from tables 3 and 7 only:

```
ARRAY LONGINT($arrTableNums;2)
$arrTableNums{1}:=3
$arrTableNums{2}:=7
VERIFY DATA FILE($StructName;$DataName;Verify records;0;"FollowScan";$arrTableNums)
```

## Example 5

Checking of specific indexes (index of field 1 of table 4 and index of fields 2 and 3 of table 5):

```
ARRAY LONGINT($arrTableNums;0) `not used but mandatory
ARRAY LONGINT($arrIndex;2;0) `2 rows (columns added later)
$arrIndex{1}{0}:=4 ` table number in element 0
APPEND TO ARRAY($arrIndex{1};1) `number of 1st field to be checked
$arrIndex{2}{0}:=5 ` table number in element 0
APPEND TO ARRAY($arrIndex{2};2) ` number of 1st field to be checked
```

```
APPEND TO ARRAY($arrIndex{2};3) ` number of 2nd field to be checked
VERIFY DATA FILE($StructName;$DataName;Verify indexes;0;"FollowScan";$arrTableNums;$arrIndex)
```

## System variables and sets

If the callback method does not exist, the verification is not carried out, an error is generated and the system variable OK is set to 0. If a log file was generated, its complete pathname is returned in the Document system variable.

# Version type

Version type -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Demo or full version, 64-bit or 32-bit version, 4D database / Merged application |

## Description

The **Version type** command returns a numeric value that denotes the type of 4D or 4D Server version that you are running. 4D provides the following predefined constants, found in the **4D Environment** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| 64 bit version | Longint | 1 | |
| Demo version | Longint | 0 | |
| Merged application | Longint | 2 | Version is an application merged with 4D Volume Desktop |

**Note:** In current versions of 4D, the demo mode is not available.

**Version type** returns a value in the form of a *bit field*; it is necessary to use bitwise operators to interpret it (see the examples).

**COMPATIBILITY NOTE:** In versions of 4D prior to 13.2, a different set of constants was available for this command; however, these constants did not handle some cases properly, so they were modified. This modification means that your code must be adapted (see example). However, if you wish to keep the previous functioning, you can just replace the constants in your existing code with their former values: 2 for 64 bit Version, 1 for Demo Version, 0 for Full Version.

## Example 1

Your 4D application contains specific code based on the version running. You can find out the execution environment using the following code:

```
If(Version type?? 64 bit Version)
 // We are in a 64-bit version
Else
 // We are in a 32-bit version
End if
```

## Example 2

This test lets you run different code depending on whether the version is a merged application or a database opened by 4D / 4D Server:

```
If(Version type?? Merged application)
 // We are in a merged application
Else
 // We are in a database executed by 4D
End if
```

# _o_ADD DATA SEGMENT

_o_ADD DATA SEGMENT
Does not require any parameters

## Description

**Compatibility note:** Starting with version 11 of 4D, data segments are no longer supported (the size of the data file is now unlimited). When it is called, this command does nothing.

# _o_DATA SEGMENT LIST

_o_DATA SEGMENT LIST ( segments )

| Parameter | Type | | Description |
|---|---|---|---|
| segments | String array | ⇦ | Long names of data segments for the database |

## Compatibility note

Since version 11 of 4D, data segments are no longer supported (the data file size is now unlimited). This command now systematically returns an array with one element containing the pathname of the data file of the database.

# 4D Write Pro

- WP CREATE BOOKMARK New 16.0
- WP DELETE BOOKMARK New 16.0
- WP EXPORT DOCUMENT
- WP EXPORT VARIABLE
- WP GET ATTRIBUTES
- WP Get bookmark range New 16.0
- WP GET BOOKMARKS New 16.0
- WP Get page count New 16.0
- WP Get paragraphs
- WP Get pictures
- WP Get range
- WP Get selection
- WP Import document
- WP INSERT BREAK New 16.0
- WP INSERT DOCUMENT New 16.0
- WP INSERT PICTURE New 16.0
- WP Is font style supported
- WP New Updated 16.0
- WP PRINT Updated 16.0
- WP RESET ATTRIBUTES
- WP SELECT
- WP SET ATTRIBUTES
- WP USE PAGE SETUP

# WP CREATE BOOKMARK

WP CREATE BOOKMARK ( rangeObj ; bkName )

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj | Object | ⇨ | 4D Write Pro range |
| bkName | String | ⇨ | Name of the bookmark to create |

## Description

The **WP CREATE BOOKMARK** command creates a new bookmark named *bkName* based upon the 4D Write Pro *rangeObj* in the range parent document.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP DELETE BOOKMARK

WP DELETE BOOKMARK ( wpDoc ; bkName )

| Parameter | Type | | Description |
|---|---|---|---|
| wpDoc | Object | ⇒ | 4D Write Pro document |
| bkName | String | ⇒ | Name of the bookmark to delete |

## Description

The **WP DELETE BOOKMARK** command removes the bookmark named *bkName* from *wpDoc*.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP EXPORT DOCUMENT

WP EXPORT DOCUMENT ( wpDoc ; filePath {; format {; options}} )

| Parameter | Type | | Description |
|---|---|---|---|
| wpDoc | Object | ➡ | 4D Write Pro variable |
| filePath | String | ➡ | Path of exported file |
| format | Longint | ➡ | Document output format |
| options | Longint | ➡ | Export options |

## Description

The **WP EXPORT DOCUMENT** command exports the *wpDoc* 4D Write Pro object to a document on disk defined by the *filePath* parameter as well as any optional parameters.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP EXPORT VARIABLE

WP EXPORT VARIABLE ( wpDoc ; destination ; format {; options} )

| Parameter | Type | | Description |
|---|---|---|---|
| wpDoc | Object | ⇒ | 4D Write Pro variable |
| destination | Text variable, BLOB variable | ⇐ | Variable to receive exported contents |
| format | Longint | ⇒ | Variable output format |
| options | Longint, String | ⇒ | Export options |

## Description

The **WP EXPORT VARIABLE** command exports the *wpDoc* 4D Write Pro object to the 4D *destination* variable in the specified *format* and with any *options* specified.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP GET ATTRIBUTES

WP GET ATTRIBUTES ( rangeObj | wpDoc ; attribName ; attribValue {; attribName2 ; attribValue2 ; ... ; attribNameN ; attribValueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj \| wpDoc | Object | ⇒ | 4D Write Pro range or document |
| attribName | String | ⇒ | Name of attribute to get |
| attribValue | String, Real, Boolean, Array | ⇐ | Current value of attribute for text range |

## Description

The **WP GET ATTRIBUTES** command returns the value of any attribute in a 4D Write Pro range or document.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP Get bookmark range

WP Get bookmark range ( wpDoc ; bkName ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| wpDoc | Object | ⇒ | 4D Write Pro variable |
| bkName | Text | ⇒ | Name of bookmark whose range you want to get |
| Function result | Object | ⇐ | Range of bookmark |

## Description

The **WP Get bookmark range** command returns a text range object (*rangeObj*) containing the range for the bookmark with the specified *bkName* in *wpDoc*.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP GET BOOKMARKS

WP GET BOOKMARKS ( wpDoc ; arrBKNames )

| Parameter | Type | | Description |
|---|---|---|---|
| wpDoc | Object | ⇒ | 4D Write Pro document |
| arrBKNames | Text array | ⇐ | Array of bookmark names |

## Description

The **WP GET BOOKMARKS** command returns an array containing the names of all bookmarks defined in *wpDoc*.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP Get page count

WP Get page count ( wpDoc ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| wpDoc | Object | ⇒ | 4D Write Pro document |
| Function result | Longint | ⇐ | Number of pages in the document |

## Description

The **WP Get page count** command returns the total number of pages defined in the *wpDoc* 4D Write Pro document.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP Get paragraphs

WP Get paragraphs ( rangeObj ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj | Object | ⇒ | Range from which to get paragraphs |
| Function result | Object | ⇐ | Range addressing only paragraphs |

## Description

The **WP Get paragraphs** command returns a specific range object that addresses only the paragraphs contained in the *rangeObj* you passed as parameter.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP Get pictures

WP Get pictures ( rangeObj ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj | Object | ⇒ | Range from which to get pictures |
| Function result | Object | ⇐ | Range object containing pictures only |

## Description

The **WP Get pictures** command returns a specific range object that addresses only the pictures contained in the *rangeObj* you passed as parameter.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP Get range

WP Get range ( wpArea ; startRange ; endRange ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| wpArea | Object | ⇒ | 4D Write Pro object variable or field |
| startRange | Longint | ⇒ | Starting offset of range in the area |
| endRange | Longint | ⇒ | Ending offset of range in the area |
| Function result | Object | ⇐ | Range object |

## Description

The **WP Get range** command returns a new text range object (*rangeObj*) containing the selection between *startRange* and *endRange* in the *wpArea* 4D Write Pro area.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP Get selection

WP Get selection ( {* ;} wpArea ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, wpArea is a form object name (string). If omitted, wpArea is an object field or variable. |
| wpArea | String, Object | ⇒ | Form object name (if * is specified) or 4D Write Pro object variable or field (if * is omitted) |
| Function result | Object | ⇐ | Range object |

## Description

The **WP Get selection** command returns a new text range object (*rangeObj*) based upon the currently selected text in the *wpArea* 4D Write Pro area.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP Import document

WP Import document ( filePath ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| filePath | String | ⇒ | Path to a 4D Write document (.4w7 or .4wt) or a 4D Write Pro document (.4wp) |
| Function result | Object | ⇐ | 4D Write Pro object |

## Description

The **WP Import document** command converts an existing 4D Write Pro or 4D Write document (.4wp, .4w7 or .4wt) to a new 4D Write Pro object.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP INSERT BREAK

WP INSERT BREAK ( rangeObj ; breakType ; mode {; rangeUpdate} )

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj | Object | ➡ | 4D Write Pro range object |
| breakType | Longint | ➡ | Type of break to insert |
| mode | Longint | ➡ | Insertion mode |
| rangeUpdate | Longint | ➡ | Range update mode |

## Description

The **WP INSERT BREAK** command inserts a new break of the *breakType* type in the *rangeObj* range according to the specified insertion *mode* and *rangeUpdate* parameter.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP INSERT DOCUMENT

WP INSERT DOCUMENT ( rangeObj ; wpDoc ; mode {; rangeUpdate} )

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj | Object | ➡ | 4D Write Pro range |
| wpDoc | Object | ➡ | 4D Write Pro document |
| mode | Longint | ➡ | Insertion mode |
| rangeUpdate | Longint | ➡ | Range update mode |

## Description

The **WP INSERT DOCUMENT** command inserts the *wpDoc* document in the *rangeObj* range according to the specified insertion *mode* and *rangeUpdate* parameter.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP INSERT PICTURE

WP INSERT PICTURE ( rangeObj ; picture ; mode {; rangeUpdate} )

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj | Object | ⇒ | Range object |
| picture | Picture, String | ⇒ | Picture field or variable, or path to picture file on disk |
| mode | Longint | ⇒ | Insertion mode |
| rangeUpdate | Longint | ⇒ | Range update mode |

## Description

The **WP INSERT PICTURE** command inserts the *picture* in *rangeObj* according to the specified insertion *mode* and *rangeUpdate* parameter.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP Is font style supported

WP Is font style supported ( rangeObj ; wpFontStyle ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj | Object | ⇒ | Range object to parse |
| wpFontStyle | Longint | ⇒ | Font style constant: wk font bold, wk font italic, wk text underline style, wk text linethrough style |
| Function result | Boolean | ⇐ | True if any part of range supports wpFontStyle; False otherwise |

## Description

The **WP Is font style supported** command returns True if the *wpFontStyle* style is supported by any part of the text in *rangeObj*.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP New

WP New {( source )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| source | String, BLOB, Object | ⇒ | String: 4D HTML source,<br>BLOB: 4D Write Blob document (.4w7/.4wt) or 4D Write Pro document (.4wp)<br>Object: a 4D Write Pro object range |
| Function result | Object | ⇐ | 4D Write Pro object |

## Description

The **WP New** command creates and returns a new 4D Write Pro object that is either empty or filled with the contents defined by *source*.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP PRINT

WP PRINT ( wpDoc {; printLayout} )

| Parameter | Type | | Description |
|---|---|---|---|
| wpDoc | Object | ⇒ | Name of 4D Write Pro document |
| printLayout | Longint | ⇒ | Print layout for 4D Write Pro document: 0 (default)=4D Write Pro layout, 1=HTML WYSIWYG |

## Description

The **WP PRINT** command launches a print job for the 4D Write Pro document specified in *wpDoc* or (64-bit versions only) adds the document to the current print job if it is called between **OPEN PRINTING JOB** and **CLOSE PRINTING JOB**.

**Note for 32-bit users:** This command is supported on 4D 32-bit versions but in this context, it cannot be used within a print job open with **OPEN PRINTING JOB**.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP RESET ATTRIBUTES

WP RESET ATTRIBUTES ( rangeObj ; attribName {; attribName2 ; ... ; attribNameN} )

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj | Object | ⇒ | 4D Write Pro range |
| attribName | String | ⇒ | Name of attribute(s) to remove |

## Description

The **WP RESET ATTRIBUTES** command allows you to reset the value of one or more attributes in the 4D Write Pro *rangeObj*.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP SELECT

WP SELECT ( {* ;} wpArea {; rangeObj}{; startRange ; endRange} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, wpArea is a form object name (string). If omitted, wpArea is an object field or variable. |
| wpArea | String, Object | ⇒ | Form object name (if * is specified) or 4D Write Pro object variable or field (if * is omitted) |
| rangeObj | Object | ⇒ | Range object to apply to create a selection |
| startRange | Longint | ⇒ | Starting offset of text range |
| endRange | Longint | ⇒ | Ending offset of text range |

## Description

The **WP SELECT** command creates a new text selection in the *wpArea* 4D Write Pro area, based upon the *rangeObj* or a new range defined by *startRange* and *endRange*.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP SET ATTRIBUTES

WP SET ATTRIBUTES ( rangeObj | wpDoc ; attribName ; attribValue {; attribName2 ; attribValue2 ; ... ; attribNameN ; attribValueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| rangeObj \| wpDoc | Object | ➡ | 4D Write Pro range or document |
| attribName | String | ➡ | Name of attribute to set |
| attribValue | String, Real, Boolean | ➡ | New attribute value |

## Description

The **WP SET ATTRIBUTES** command allows you to set the value of any attribute in a 4D Write Pro range or document.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# WP USE PAGE SETUP

WP USE PAGE SETUP ( wpDoc )

| Parameter | Type | | Description |
|---|---|---|---|
| wpDoc | Object | ⇨ | Name of 4D Write Pro document |

## Description

The **WP USE PAGE SETUP** command modifies the current printer page settings based on the page size and orientation attributes for the 4D Write Pro document passed in *wpDoc*.

For more information, refer to the description of this command in the **4D Write Pro Language** chapter of the *4D Write Pro Reference* manual.

# Arrays

- Arrays
- Creating Arrays
- Arrays and Form Objects
- Arrays and the 4D Language
- Arrays and Pointers
- Using the element zero of an array
- Two-dimensional Arrays
- Arrays and Memory
- APPEND TO ARRAY
- ARRAY BLOB
- ARRAY BOOLEAN
- ARRAY DATE
- ARRAY INTEGER
- ARRAY LONGINT
- ARRAY OBJECT
- ARRAY PICTURE
- ARRAY POINTER
- ARRAY REAL
- ARRAY TEXT
- ARRAY TIME
- ARRAY TO LIST
- ARRAY TO SELECTION
- BOOLEAN ARRAY FROM SET
- COPY ARRAY
- Count in array
- DELETE FROM ARRAY
- DISTINCT ATTRIBUTE PATHS New 16.0
- DISTINCT ATTRIBUTE VALUES New 16.0
- DISTINCT VALUES
- Find in array
- Find in sorted array
- INSERT IN ARRAY
- LIST TO ARRAY
- LONGINT ARRAY FROM SELECTION
- MULTI SORT ARRAY
- SELECTION RANGE TO ARRAY
- SELECTION TO ARRAY
- Size of array
- SORT ARRAY
- TEXT TO ARRAY
- *_o_ARRAY STRING*

# Arrays

An **array** is an ordered series of variables of the same type. Each variable is called an **element** of the array. The **size** of an array is the number of elements it holds. An array is given its size when it is created; you can then resize it as many times as needed by adding, inserting, or deleting elements, or by resizing the array using the same command used to create it.

You create an array with one of the array declaration commands. For details, see the **Creating Arrays** section.

Elements are numbered from **1 to N**, where N is the size of the array. An array always has an **element zero** that you can access just like any other element of the array, but this element is not shown when an array is present in a form. Although the element zero is not shown when an array supports a form object, there is no restriction in using it with the language. For more information about the element zero, see the **Using the element zero of an array** section.

Arrays are 4D variables. Like any variable, an array has a scope and follows the rules of the 4D language, though with some unique differences. For more information, see the **Arrays and the 4D Language** and **Arrays and Pointers** sections.

Arrays are language objects; you can create and use arrays that will never appear on the screen. Arrays are also user interface objects. For more information about the interaction between arrays and form objects, see the **Arrays and Form Objects** section.

Arrays are designed to hold reasonable amounts of data for a short period of time. However, because arrays are held in memory, they are easy to handle and quick to manipulate. For details, see the **Arrays and Memory** section.

## Creating Arrays

You create an array with one of the array declaration commands described in this chapter. The following table lists the array declaration commands:

| Command | Creates or resizes an array of |
|---|---|
| **ARRAY INTEGER** | 2-byte Integer values |
| **ARRAY LONGINT** | 4-byte Integer values |
| **ARRAY REAL** | Real values |
| **ARRAY TEXT** | Text values (up to 2 GB of text per element) (*) |
| **_o_ARRAY STRING** | Text values (obsolete) (*) |
| **ARRAY DATE** | Date values |
| **ARRAY BOOLEAN** | Boolean values |
| **ARRAY PICTURE** | Pictures values |
| **ARRAY POINTER** | Pointer values |
| **ARRAY OBJECT** | Language objects |
| **ARRAY BLOB** | BLOBs |
| **ARRAY TIME** | Times |

Each array declaration command can create or resize one-dimensional or two-dimensional arrays. For more information about two-dimensional arrays, see the **Two-dimensional Arrays** section.

(*) There is no difference between Text and String arrays. The *strLen* parameter of the **_o_ARRAY STRING** command is ignored. It is recommended to use Text arrays. The **_o_ARRAY STRING** command is only kept for compatibility reasons.

The following line of code creates (declares) an Integer array of 10 elements:

```
ARRAY INTEGER(aiAnArray;10)
```

Then, the following code resizes that same array to 20 elements:

```
ARRAY INTEGER(aiAnArray;20)
```

Then, the following code resizes that same array to no elements:

```
ARRAY INTEGER(aiAnArray;0)
```

You reference the elements in an array by using curly braces ({…}). A number is used within the braces to address a particular element; this number is called the **element number**. The following lines put five names into the array called *atNames* and then display them in alert windows:

```
ARRAY TEXT(atNames;5)
atNames{1}:="Richard"
atNames{2}:="Sarah"
atNames{3}:="Sam"
atNames{4}:="Jane"
atNames{5}:="John"
For($vlElem;1;5)
   ALERT("The element #"+String($vlElem)+" is equal to: "+atNames{$vlElem})
End for
```

Note the syntax *atNames{$vlElem}*. Rather than specifying a numeric literal such as *atNames{3}*, you can use a numeric variable to indicate which element of an array you are addressing.

Using the iteration provided by a loop structure (**For...End for**, **Repeat...Until** or **While...End while**), compact pieces of code can address all or part of the elements in an array.

## Arrays and other areas of the 4D language

There are other 4D commands that can create and work with arrays. More particularly:

- To work with arrays and selection of records, use the commands **SELECTION RANGE TO ARRAY**, **SELECTION TO ARRAY**, **ARRAY TO SELECTION** and **DISTINCT VALUES**.
- Objects of the List box type are based on arrays; several commands of the “List box” theme work with arrays, for instance **LISTBOX INSERT ROWS**.
- You can create graphs and charts on series of values stored in tables and arrays. For more information, see the **GRAPH** command.
- The **LIST TO ARRAY** and **ARRAY TO LIST** commands.
- Many commands can build arrays in one call, for example: **FONT LIST**, **WINDOW LIST**, **VOLUME LIST**, **FOLDER LIST**, **DOCUMENT LIST**, **GET SERIAL PORT MAPPING**, **SAX GET XML ELEMENT**, etc.

# Arrays and Form Objects

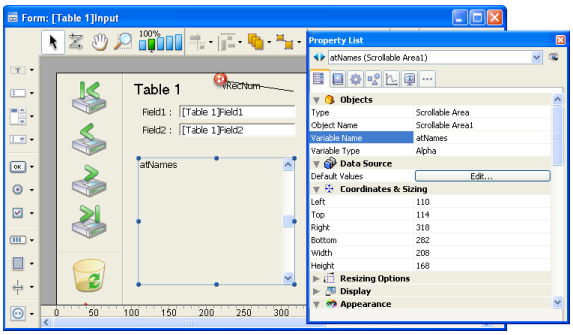
Arrays are language objects—you can create and use arrays that will never appear on the screen. However, arrays are also user interface objects. The following types of **Form Objects** are supported by arrays:

- Pop-up/Drop-down List
- Combo Box
- Scrollable Area (obsolete starting with 4D v13)
- Tab Control
- List box

While you can predefine these objects in the Design Environment Form Editor using the Default Values button of the Property List window (except for the List box) , you can also define them programmatically using the arrays commands. In both cases, the form object is supported by an array created by you or 4D.

When using these objects, you can detect which item within the object has been selected (or clicked) by testing the array for its **selected element**. Conversely, you can select a particular item within the object by setting the selected element for the array.

When an array is used to support a form object, it has then a dual nature; it is both a language object and a user interface object. For example, when designing a form, you create a scrollable area:




The name of the associated variable, in this case *atNames*, is the name of the array you use for creating and handling the scrollable area.

**Notes**:

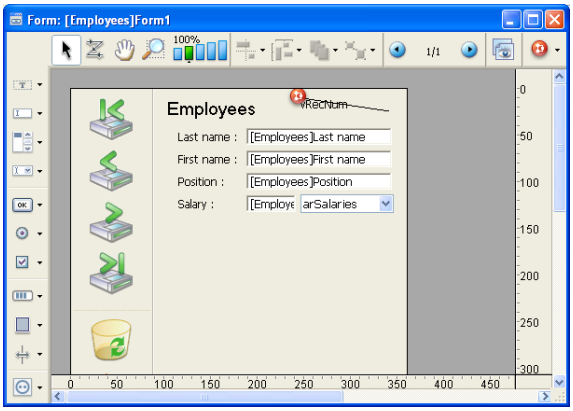
- The **selected element** of the array is stored internally in an Integer type variable. As a result, it is not possible to use it with arrays containing more than 32,767 elements (however, such a large number of elements is also not suitable for display as a form object).
- You cannot display two-dimensional arrays or pointer arrays.
- The management of **List box** type objects (which may contain several arrays) entails many specific aspects. These particularities are covered in the **Managing List Box Objects** section.

## Example: Creating a drop-down list

The following example shows how to fill an array and display it in a drop-down list. An array *arSalaries* is created using the **ARRAY REAL** command. It contains all the standard salaries paid to people in a company. When the user chooses an element from the drop-down list, the *[Employees]Salary* field is assigned the value chosen.

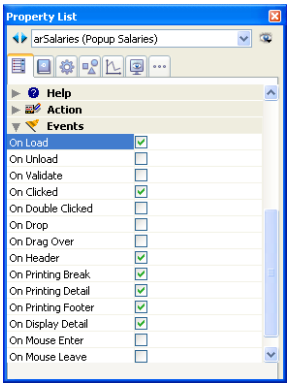
### Create the arSalaries drop-down list on a form

Create a drop-down list and name it *arSalaries*. The name of the drop-down list should be the same as the name of the array.

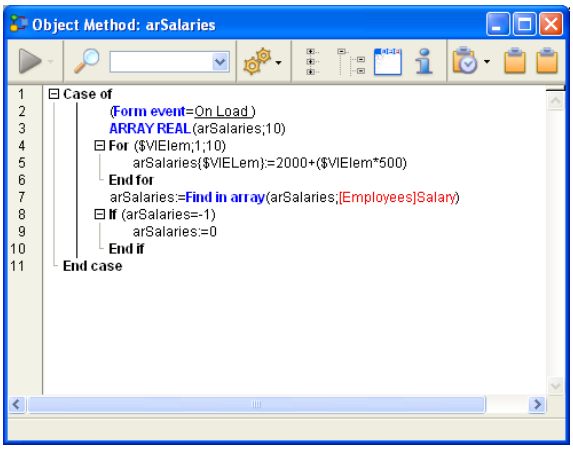



### Initializing the array

Initialize the array *arSalaries* using the On Load event for the object. To do so, remember to enable that event in the **Property List** window, as shown:




Click the **Object Method** button and create the method, as follows:




The lines:

```
ARRAY REAL(arSalaries;10)
For($vlElem;1;10)
   arSalaries{$vlElem}:=2000+($vlElem*500)
End for
```

create the numeric array 2500, 3000... 7000, corresponding to the annual salaries $30,000 up to $84,000, before tax.

The lines:

```
arSalaries:=Find in array(arSalaries;[Employees]Salary)
If(arSalaries=-1)
   arSalaries:=0
End if
```

handle both the creation of a new record or the modification of existing record.

- If you create a new record, the field *[Employees]Salary* is initially equal to zero. In this case, **Find in array** does not find the value in the array and returns -1. The test *If (arSalaries=-1)* resets *arSalaries* to zero, indicating that no element is selected in the drop-down list.

- If you modify an existing record, **Find in array** retrieves the value in the array and sets the selected element of the drop-down list to the current value of the field. If the value for a particular employee is not in the list, the test *If (arSalaries=-1)* deselects any element in the list.

**Note:** For more information about the **array selected element**, read the next section.

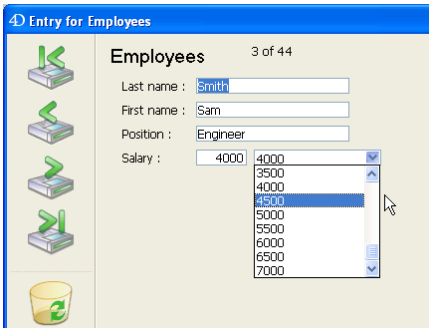
### Reporting the selected value to the [Employees]Salary field

To report the value selected from the drop-down list *arSalaries*, you just need to handle the On Clicked event to the object. The element number of the selected element is the value of the array *arSalaries* itself. Therefore, the expression *arSalaries{arSalaries}* returns the value chosen in the drop-down list.

Complete the method for the object *arSalaries* as follows:

```
Case of
   :(Form event=On Load)
      ARRAY REAL(arSalaries;10)
      For($vlElem;1;10)
         arSalaries{$vlElem}:=2000+($vlElem*500)
      End for
      arSalaries:=Find in array(arSalaries;[Employees]Salary)
      If(arSalaries=-1)
         arSalaries:=0
      End if
   :(Form event=On Clicked)
      [Employees]Salary:=arSalaries{arSalaries}
End case
```

The drop-down list looks like this:




The following section describes the common and basic operations you will perform on arrays while using them as form objects.

## Getting the size of the array

You can obtain the current size of the array by using the **Size of array** command. Using the previous example, the following line of code would display 5:

```
ALERT("The size of the array atNames is: "+String(Size of array(atNames)))
```

## Reordering the elements of the array

You can reorder the elements of the array using the **SORT ARRAY** command or of several arrays using the **MULTI SORT ARRAY** command. Using the previous example, and provided the array is shown as a scrollable area:

a. Initially, the area would look like the list on the left.

b. After the execution of the following line of code:

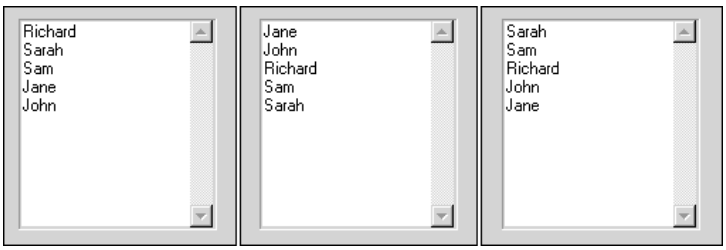
```
SORT ARRAY(atNames;>)
```

the area would look like the list in the middle.

c. After the execution of the following line of code:

```
SORT ARRAY(atNames;<)
```

the area would look like the list on the right.




## Adding or deleting elements

You can add, insert, or delete elements using the commands **APPEND TO ARRAY**, **INSERT IN ARRAY** and **DELETE FROM ARRAY**.

## Handling clicks in the array: testing the selected element

Using the previous example, and provided the array is shown as a scrollable area, you can handle clicks in this area as follows:

```
 //atNames scrollable area object method
Case of
   :(Form event=On Load)
 //Initialize the array (as shown further above)
      ARRAY TEXT(atNames;5)
 // ...
   :(Form event=On Unload)
 //We no longer need the array
      CLEAR VARIABLE(atNames)

   :(Form event=On Clicked)
      If(atNames#0)
         vtInfo:="You clicked on: "+atNames{atNames}
      End if
   :(Form event=On Double Clicked)
      If(atNames#0)
         ALERT("You double clicked on: "+atNames{atNames}
      End if
End case
```

**Note**: The events must be activated in the properties of the object.

While the syntax *atNames{$vlElem}* allows you to work with a particular element of the array, the syntax *atNames* returns the element number of the selected element within the array. Thus, the syntax *atNames{atNames}* means "the value of the selected element in the array *atNames*." If no element is selected, *atNames* is equal to 0 (zero), so the test **If (atNames#0)** detects whether or not an element is actually selected.

## Setting the selected element

In a similar fashion, you can programmatically change the selected element by assigning a value to the array.

### Examples

```
 //Selects the first element (if the array is not empty)
atNames:=1

 //Selects the last element (if the array is not empty)
atNames:=Size of array(atNames)

 //Deselects the selected element (if any) then no element is selected
atNames:=0

If((0<atNames)&(atNames<Size of array(atNames))
 //If possible, selects the next element to the selected element
   atNames:=atNames+1
End if
```

```
If(1<atNames)
 //If possible, selects the previous element to the selected element
   atNames:=atNames-1
End if
```

## Looking for a value in the array

The **Find in array** command searches for a particular value within an array. Using the previous example, the following code will select the element whose value is “Richard,” if that is what is entered in the request dialog box:

```
$vsName:=Request("Enter the first name:")
If(OK=1)
   $vlElem:=Find in array(atNames;$vsName)
   If($vlElem>0)
      atNames:=$vlElem
   Else
      ALERT("This is no "+$vsName+" in that list of first names.")
   End if
End if
```

Pop-up menus, drop-down lists, scrollable areas, and tab controls can be usually handled in the same manner. Obviously, no additional code is required to redraw objects on the screen each time you change the value of an element, or add or delete elements.

**Note**: To create and use tab controls with icons and enabled and disabled tabs, you must use a hierarchical list as the supporting object for the tab control. For more information, see the example for the **Count tasks** command.

## Handling combo boxes

While you can handle pop-up menus, drop-down lists, scrollable areas, and tab controls with the algorithms described in the previous section, you must handle combo boxes differently.

A combo box is actually a text enterable area to which is attached a list of values (the elements from the array). The user can pick a value from this list, and then edit the text. So, in a combo box, the notion of selected element does not apply.

With combo boxes, there is never a selected element. Each time the user selects one of the values attached to the area, that value is put into the element zero of the array. Then, if the user edits the text, the value modified by the user is also put into that element zero.

### Example

```
  ` asColors Combo Box object method
Case of
   :(Form event=On Load)
      ARRAY STRING(31;asColors;3)
      asColors{1}:="Blue"
      asColors{2}:="White"
      asColors{3}:="Red"
   :(Form event=On Clicked)
      If(asColors{0}#"")
  ` The object automatically changes its value
  ` Using the On Clicked event with a Combo Box
  ` is required only when additional actions must be taken
      End if
   :(Form event=On Data Change)
  ` Find in array ignores element 0, so returns -1 or >0
      If(Find in array(asColors;asColors{0})<0)
  ` Entered value is not one the values attached to the object
  ` Add the value to the list for next time
         APPEND TO ARRAY(asColors;asColors{0})
      Else
  ` Entered value is among the values attached to the object
      End if
End case
```

# Arrays and the 4D Language

Arrays are 4D variables. Like any variable, an array has a scope and follows the rules of the 4D language, though with some unique differences.

## Local, process and interprocess arrays

You can create and work with local, process, and interprocess arrays. Examples:

```
ARRAY INTEGER($aiCodes;100) ` Creates local array of 100 2-byte Integer values
ARRAY INTEGER(aiCodes;100) ` Creates process array of 100 2-byte Integer values
ARRAY INTEGER(◊aiCodes;100) ` Creates interprocess array of 100 2-byte Integer values
```

The scope of these arrays is identical to the scope of other local, process, and interprocess variables:

### Local arrays

A local array is declared when the name of the array starts with a dollar sign ($).

The scope of a local array is the method in which it is created. The array is cleared when the method ends. Local arrays with the same name in two different methods can have different types, because they are actually two different variables with different scopes.

When you create a local array within a form method, within an object method, within or a project method called as subroutine by the two previous type of method, the array is created and cleared each time the form or object method is invoked. In other words, the array is created and cleared for each form event. Consequently, you cannot use local arrays in forms, neither for display nor printing.

As with local variables, it is a good idea to use local arrays whenever possible. In doing so, you tend to minimize the amount of memory necessary for running your application.

### Process arrays

A process array is declared when the name of the array starts with a letter.

The scope of a process array is the process in which it is created. The array is cleared when the process ends or is aborted. A process array automatically has one instance created per process. Therefore, the array is of the same type throughout the processes. However, its contents are particular to each process.

### Interprocess arrays

An interprocess array is declared when the name of the array starts with <> (on Windows and Macintosh) or with the diamond sign, Option-Shift-V on a US keyboard (on Macintosh only).

The scope of an interprocess array consists of all processes during a working session. They should be used only to share data and transfer information between processes.

**Tip**: When you know in advance that an interprocess array will be accessed by several processes that could possible conflict, protect the access to that array with a semaphore. For more information, see the example for the **Semaphore** command.

**Note**: You can use process and interprocess arrays in forms to create form objects such as scrollable areas, drop-down lists, and so on.

## Passing an Array as parameter

You can pass an array as parameter to a 4D command or to the routine of a 4D Plug-in. On the other hand, you cannot pass an array as parameter to a user method. The alternative is to pass a pointer to the array as parameter to the method. For details, see the **Arrays and Pointers** section.

## Assigning and array to another array

Unlike text or string variables, you cannot assign one array to another. To copy (assign) an array to another one, use **COPY ARRAY**.

## Arrays and Pointers

---

You can pass an array as parameter to a 4D command or to the routine of a 4D Plug-in. On the other hand, you cannot pass an array as parameter to a user method. The alternative is to pass a pointer to the array as parameter to the method.

You can pass process and interprocess, process or local arrays as parameters.

Here are some examples.

- Given this example:

```
If((0<atNames)&(atNames<Size of array(atNames))
  ` If possible, selects the next element to the selected element
   atNames:=atNames+1
End if
```

If you need to do the same thing for 50 different arrays in various forms, you can avoid writing the same thing 50 times, by using the following project method:

```
  ` SELECT NEXT ELEMENT project method
  ` SELECT NEXT ELEMENT ( Pointer )
  ` SELECT NEXT ELEMENT ( -> Array )

C_POINTER($1)

If((0<$1->)&($1-><Size of array($1->))
   $1->:=$1->+1 ` If possible, selects the next element to the selected element
End if
```

Then, you can write:

```
SELECT NEXT ELEMENT(->atNames)
  ` ...
SELECT NEXT ELEMENT(->asZipCodes)
  ` ...
SELECT NEXT ELEMENT(->alRecordIDs)
  ` ... and so on
```

- The following project method returns the sum of all the elements of a numeric array (Integer, Long Integer, or real):

```
  ` Array sum
  ` Array sum ( Pointer )
  ` Array sum ( -> Array )

C_REAL($0)

$0:=0
For($vlElem;1;Size of array($1->))
   $0:=$0+$1->{$vlElem}
End for
```

**Note:** Since 4D v13, you can just use the **Sum** function to calculate the sum of the elements of a number array.

Then, you can write:

```
$vlSum:=Array sum(->arSalaries)
  ` ...
$vlSum:=Array sum(->aiDefectCounts)
  ` ..
$vlSum:=Array sum(->alPopulations)
```

- The following project method capitalizes of all the elements of a string or text array:

```
  ` CAPITALIZE ARRAY
  ` CAPITALIZE ARRAY ( Pointer )
  ` CAPITALIZE ARRAY ( -> Array )

For($vlElem;1;Size of array($1->))
   If($1->{$vlElem}#"")
      $1->{$vlElem}:=Uppercase($1->{$vlElem}≤1≥)+Lowercase(Substring($1->{$vlElem};2))
   End if
End for
```

Then, you can write:

```
CAPITALIZE ARRAY(->atSubjects)
  ` ...
CAPITALIZE ARRAY(->asLastNames)
```

The combination of arrays, pointers, and looping structures, such as **For...End for**, allows you to write many useful small project methods for handling arrays.

## Using the element zero of an array

---

An array always has an element zero. While element zero is not shown when an array supports a form object, there is no restriction(*) in using it with the language.

One example of the use of element zero is the case of the combo box discussed in the **Arrays and Form Objects** section.

Here is another example: you want to execute an action only when you click on an element other than the previously selected element. To do this, you must keep track of each selected element. One way to do this is to use a process variable in which you maintain the element number of the selected element. Another way is to use the element zero of the array:

```
  ` atNames scrollable area object method
Case of
   :(Form event=On Load)
  ` Initialize the array (as shown further above)
      ARRAY TEXT(atNames;5)
  ` ...
  ` Initialize the element zero with the number
  ` of the current selected element in its string form
  ` Here you start with no selected element
      atNames{0}:="0"

   :(Form event=On Unload)
  ` We no longer need the array
      CLEAR VARIABLE(atNames)

   :(Form event=On Clicked)
      If(atNames#0)
         If(atNames#Num(atNames{0}))
            vtInfo:="You clicked on: "+atNames{atNames}+" and it was not selected before."
            atNames{0}:=String(atNames)
         End if
      End if
   :(Form event=On Double Clicked)
      If(atNames#0)
         ALERT("You double clicked on: "+atNames{atNames}
      End if
End case
```

(*) However, there is one exception: in an array type **List Box**, the zero element is used internally to store the previous value of an element being edited, so it is not possible to use it in this particular context.

## Two-dimensional Arrays

---

Each of the array declaration commands can create or resize one-dimensional or two-dimensional arrays. Example:

```
ARRAY TEXT(atTopics;100;50) ` Creates a text array composed of 100 rows of 50 columns
```

Two-dimensional arrays are essentially language objects; you can neither display nor print them.

In the previous example:

- *atTopics* is a two-dimensional array
- *atTopics{8}{5}* is the 5th element (5th column...) of the 8th row
- *atTopics{20}* is the 20th row and is itself a one-dimensional array
- **Size of array***(atTopics)* returns 100, which is the number of rows
- **Size of array***(atTopics{17})* returns 50, which the number of columns for the 17th row

In the following example, a pointer to each field of each table in the database is stored in a two-dimensional array:

```
C_LONGINT($vlLastTable;$vlLastField)
C_LONGINT($vlFieldNumber)
 ` Create as many rows (empty and without columns) as there are tables
$vlLastTable:=Get last table number
ARRAY POINTER(<>apFields;$vlLastTable;0) `2D array with X rows and zero columns
 ` For each table
For($vlTable;1;$vlLastTable)
   If(Is table number valid($vlTable))
      $vlLastField:=Get last field number($vlTable)
 ` Give value of elements
      $vlColumnNumber:=0
      For($vlField;1;$vlLastField)
         If(Is field number valid($vlTable;$vlField))
            $vlColumnNumber:=$vlColumnNumber+1
 `Insert a column in a row of the table underway
            INSERT IN ARRAY(<>apFields{$vlTable};$vlColumnNumber;1)
 `Assign the "cell" with the pointer
            <>apFields{$vlTable}{$vlColumnNumber}:=Field($vlTable;$vlField)
         End if
      End for
   End if
End for
```

Provided that this two-dimensional array has been initialized, you can obtain the pointers to the fields for a particular table in the following way:

```
 ` Get the pointers to the fields for the table currently displayed at the screen:
COPY ARRAY(◊apFields{Table(Current form table)};$apTheFieldsIamWorkingOn)
 ` Initialize Boolean and Date fields
For($vlElem;1;Size of array($apTheFieldsIamWorkingOn))
   Case of
      :(Type($apTheFieldsIamWorkingOn{$vlElem}->)=Is date)
         $apTheFieldsIamWorkingOn{$vlElem}->:=Current date
      :(Type($apTheFieldsIamWorkingOn{$vlElem}->)=Is Boolean)
         $apTheFieldsIamWorkingOn{$vlElem}->:=True
   End case
End for
```

**Note**: As this example suggests, rows of a two-dimensional arrays can be the same size or different sizes.

## Arrays and Memory

---

Unlike the data you store on disk using tables and records, an array is always held in memory in its entirety.

For example, if all US zip codes were entered in the *[Zip Codes]* table, it would contain about 100,000 records. In addition, that table would include several fields: the zip code itself and the corresponding city, county, and state. If you select only the zip codes from California, the 4D database engine creates the corresponding selection of records within the *[Zip Codes]* table, and then loads the records only when they are needed (i.e., when they are displayed or printed). In order words, you work with an ordered series of values (of the same type for each field) that is partially loaded from the disk into the memory by the database engine of 4D.

Doing the same thing with arrays would be prohibitive for the following reasons:

- In order to maintain the four information types (zip code, city, county, state), you would have to maintain four large arrays in memory.
- Because an array is always held in memory in its entirety, you would have to keep all the zip codes information in memory throughout the whole working session, even though the data is not always in use.
- Again, because an array is always held in memory in its entirety, each time the database is started and then quit, the four arrays would have to be loaded and then saved on the disk, even though the data is not used or modified during the working session.

**Conclusion**: Arrays are intended to hold reasonable amounts of data for a short period of time. On the other hand, because arrays are held in memory, they are easy to handle and quick to manipulate.

However, in some circumstances, you may need to work with arrays holding hundreds or thousands of elements. The following table lists the formulas used to calculate the amount of memory used for each array type:

| Array Type | Formula for determining Memory Usage in Bytes |
|---|---|
| Boolean | (31+number of elements)¥8 |
| Date | (1+number of elements) * 6 |
| String | (1+number of elements) * (Sum of the size of each text) |
| Integer | (1+number of elements) * 2 |
| Long Integer | (1+number of elements) * 4 |
| Picture | (1+number of elements) * 4 + Sum of the size of each picture |
| Pointer | (1+number of elements) * 16 |
| Real | (1+number of elements) * 8 |
| Text | (1+number of elements) * (Sum of the size of each text) |
| Two-dimensional | (1+number of elements) * 12 + Sum of the size of each array |

**Notes**:

- The size of a text in memory is calculated using this formula: ((Length + 1) * 2)
- A few additional bytes are required to keep track of the selected element, the number of elements, and the array itself.

When working with very large arrays, the best way to handle full memory situations is to surround the creation of the arrays with an **ON ERR CALL** project method. Example:

```
  ` You are going to run a batch operation the whole night
  ` that requires the creation of large arrays. Instead of risking
  ` occurrences of errors in the middle of the night, put
  ` the creation of the arrays at the beginning of the operation
  ` and test the errors at this moment:
gError:=0 ` Assume no error
ON ERR CALL("ERROR HANDLING") ` Install a method for catching errors
ARRAY STRING(63;asThisArray;50000) ` Roughly 3125K in ASCII mode
ARRAY REAL(arThisAnotherArray;50000) ` 488K
ON ERR CALL("") ` No longer need to catch errors
If(gError=0)
  ` The arrays could be created
  ` and let's pursue the operation
Else
```

```
      ALERT("This operation requires more memory!")
   End if
    ` Whatever the case, we no longer need the arrays
   CLEAR VARIABLE(asThisArray)
   CLEAR VARIABLE(arThisAnotherArray)
```

The ***ERROR HANDLING*** project method is listed here:

```
    ` ERROR HANDLING project method
   gError:=Error ` Just return the error code
```

# APPEND TO ARRAY

APPEND TO ARRAY ( array ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ➡ | Array to which an element will be appended |
| value | Expression | ➡ | Value to append |

## Description

The **APPEND TO ARRAY** command adds a new element at the end of *array* and assigns *value* to the element. In interpreted mode, if *array* does not exist, the command creates it with regard to the type of *value*.

This command works with all kind of arrays: string, number, Boolean, date, pointer and picture.

The type of *value* must match the array type, otherwise the syntax error 54 “Argument types are incompatible” is generated. The following values will, however, be accepted:

- A string *array* (Text or String) accepts any *value* of the Text or String type.
- A number *array* (Integer, Longint or Real) accepts any *value* of the Integer, Longint, Real or Time type.

## Example

The following code:

```
INSERT IN ARRAY($myarray;Size of array($myarray)+1)
$myarray{Size of array($myarray)}:=$myvalue
```

... can be replaced with:

```
APPEND TO ARRAY($myarray;$myvalue)
```

# ARRAY BLOB

ARRAY BLOB ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ➡ | Name of array |
| size | Longint | ➡ | Number of array elements or Number of arrays if size2 is specified |
| size2 | Longint | ➡ | Number of 2D array elements |

## Description

The **ARRAY BLOB** command creates and/or resizes an array of Blob type elements in memory.

The *arrayName* parameter is the name of the array.

The *size* parameter is the number of array elements.

The *size2* parameter is optional. If you pass it, this command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* the number of columns in each array. Each row in a two-dimensional array can be processed both as an element and an array. This means that when you work with the first dimension of a two-dimensional array, you can insert and remove entire arrays using other commands in this theme.

When you apply the **ARRAY BLOB** command to an existing array:

- If you enlarge its size, existing elements are not changed and new elements are initialized to an empty BLOB (**BLOB size**= 0).
- If you reduce its size, elements at the "bottom" of the array are deleted and lost.

## Example 1

This example creates a process array containing 100 BLOB-type elements:

```
ARRAY BLOB(arrBlob;100)
```

## Example 2

This example creates a local array of 100 rows each containing 50 BLOB-type elements:

```
ARRAY BLOB($arrBlob;100;50)
```

## Example 3

This example creates a local array of 100 rows each containing 50 BLOB-type elements. The *$vByteValue* variable receives the 10th byte of the BLOB placed in the 7th column and the 5th row of the BLOB array:

```
C_INTEGER($vByteValue)
ARRAY BLOB($arrValues;100;50)
...
$vByteValue:=$arrValues{5}{7}{9}
```

# ARRAY BOOLEAN

ARRAY BOOLEAN ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ⇒ | Name of the array |
| size | Longint | ⇒ | Number of elements in the array or Number of rows if size2 is specified |
| size2 | Longint | ⇒ | Number of columns in a two-dimensional array |

## Description

The **ARRAY BOOLEAN** command creates and/or resizes an array of *Boolean* elements in memory.

The *arrayName* parameter is the name of the array.

The *size* parameter is the number of elements in the array.

The *size2* parameter is optional; if *size2* is specified, the command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying **ARRAY BOOLEAN** to an existing array:

- If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to False.
- If you reduce the array size, the last elements deleted from the array are lost.

**Tip:** In some contexts, an alternative to using Boolean arrays is using an Integer array where each element “means true” if different from zero and “means false” if equal to zero.

## Example 1

This example creates a process array of 100 *Boolean* elements:

```
ARRAY BOOLEAN(abValues;100)
```

## Example 2

This example creates a local array of 100 rows of 50 *Boolean* elements:

```
ARRAY BOOLEAN($abValues;100;50)
```

## Example 3

This example creates an interprocess array of 50 *Boolean* elements and sets each even element to True:

```
ARRAY BOOLEAN(◊abValues;50)
For($vlElem;1;50)
   ◊abValues{$vlElem}:=(($vlElem%2)=0)
End for
```

# ARRAY DATE

ARRAY DATE ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ➡ | Name of the array |
| size | Longint | ➡ | Number of elements in the array or Number of rows if size2 is specified |
| size2 | Longint | ➡ | Number of columns in a two-dimensional array |

## Description

The **ARRAY DATE** command creates and/or resizes an array of *Date* elements in memory.

- The *arrayName* parameter is the name of the array.
- The *size* parameter is the number of elements in the array.
- The *size2* parameter is optional; if *size2* is specified, the command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying to an existing array:

- If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to the null date (!00/00/00!).
- If you reduce the array size, the last elements deleted from the array are lost.

## Example 1

This example creates a process array of 100 Date elements:

```
ARRAY DATE(adValues;100)
```

## Example 2

This example creates a local array of 100 rows of 50 Date elements:

```
ARRAY DATE($adValues;100;50)
```

## Example 3

This example creates an interprocess array of 50 Date elements, and sets each element to the current date plus a number of days equal to the element number:

```
ARRAY DATE(◊adValues;50)
For($vlElem;1;50)
   ◊adValues{$vlElem}:=Current date+$vlElem
End for
```

# ARRAY INTEGER

ARRAY INTEGER ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ⇒ | Name of the array |
| size | Longint | ⇒ | Number of elements in the array or Number of rows if size2 is specified |
| size2 | Longint | ⇒ | Number of columns in a two-dimensional array |

## Description

The **ARRAY INTEGER** command creates and/or resizes an array of 2-byte *Integer* elements in memory.

- The *arrayName* parameter is the name of the array.
- The *size* parameter is the number of elements in the array.
- The *size2* parameter is optional; if *size2* is specified, the command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying **ARRAY INTEGER** to an existing array:

- If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to 0.
- If you reduce the array size, the last elements deleted from the array are lost.

## Example 1

This example creates a process array of 100 2-byte *Integer* elements:

```
ARRAY INTEGER(aiValues;100)
```

## Example 2

This example creates a local array of 100 rows of 50 2-byte *Integer* elements:

```
ARRAY INTEGER($aiValues;100;50)
```

## Example 3

This example creates an interprocess array of 50 2-byte *Integer* elements, and sets each element to its element number:

```
ARRAY INTEGER(◊aiValues;50)
For($vlElem;1;50)
   ◊aiValues{$vlElem}:=$vlElem
End for
```

# ARRAY LONGINT

ARRAY LONGINT ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ⇒ | Name of the array |
| size | Longint | ⇒ | Number of elements in the array or Number of rows if size2 is specified |
| size2 | Longint | ⇒ | Number of columns in a two-dimensional array |

## Description

The **ARRAY LONGINT** command creates and/or resizes an array of 4-byte *Longint* elements in memory.

- The *arrayName* parameter is the name of the array.
- The *size* parameter is the number of elements in the array.
- The *size2* parameter is optional; if *size2* is specified, the command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

When applying **ARRAY LONGINT** to an existing array:

- If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to 0.
- If you reduce the array size, the last elements deleted from the array are lost.

## Example 1

This example creates a process array of 100 4-byte Long Integer elements:

```
ARRAY LONGINT(alValues;100)
```

## Example 2

This example creates a local array of 100 rows of 50 4-byte Long Integer elements:

```
ARRAY LONGINT($alValues;100;50)
```

## Example 3

This example creates an interprocess array of 50 4-byte Long Integer elements and sets each element to its element number:

```
ARRAY LONGINT(◊alValues;50)
For($vlElem;1;50)
   ◊alValues{$vlElem}:=$vlElem
End for
```

# ARRAY OBJECT

ARRAY OBJECT ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ➡ | Name of array |
| size | Longint | ➡ | Number of array elements or Number of arrays if size2 is specified |
| size2 | Longint | ➡ | Number of 2D array elements |

## Description

The **ARRAY OBJECT** command creates and/or resizes an array of language Object type elements in memory.

The *arrayName* parameter is the name of the array. You can use any name that conforms to 4D conventions.

The *size* parameter is the number of array elements.

The *size2* parameter is optional. If you pass it, this command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* the number of columns in each array. Each row in a two-dimensional array can be processed both as an element and an array. This means that when you work with the first dimension of a two-dimensional array, you can insert and remove entire arrays using other commands in the "Arrays" theme.

When you apply the **ARRAY OBJECT** command to an existing array:

- If you enlarge its size, existing elements are not changed and new elements are undefined. You can test whether an element is defined using the **OB Is defined** command.
- If you reduce its size, elements at the "bottom" of the array are deleted and lost.

## Example 1

Creation of a process array of 100 Object-type elements:

```
ARRAY OBJECT(arrObjects;100)
```

## Example 2

Creation of a local array of 100 rows each containing 50 Object-type elements:

```
ARRAY OBJECT($arrObjects;100;50)
```

## Example 3

Creation and filling of a local object array:

```
C_OBJECT($Children;$ref_richard;$ref_susan;$ref_james)
ARRAY OBJECT($arrayChildren;0)
OB SET($ref_richard;"name";"Richard";"age";7)
APPEND TO ARRAY($arrayChildren;$ref_richard)
OB SET($ref_susan;"name";"Susan";"age";4)
APPEND TO ARRAY($arrayChildren;$ref_susan)
OB SET($ref_james;"name";"James";"age";3)
APPEND TO ARRAY($arrayChildren;$ref_james)
 // $arrayChildren{1} -> {"name":"Richard","age":7}
 // $arrayChildren{2} -> {"name":"Susan","age":4}
 // $arrayChildren{3} -> {"name":"James","age":3}
```

# ARRAY PICTURE

ARRAY PICTURE ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ⇒ | Name of the array |
| size | Longint | ⇒ | Number of elements in the array, or Number of rows if size2 is specified |
| size2 | Longint | ⇒ | Number of columns in a two-dimensional array |

## Description

The **ARRAY PICTURE** command creates and/or resizes an array of *Picture* elements in memory.

- The *arrayName* parameter is the name of the array.
- The *size* parameter is the number of elements in the array.The *size2* parameter is optional; if *size2* is specified, the command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying to an existing array:

- If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to empty pictures. This means that **Picture size** applied to one of these elements will return *0*.
- If you reduce the array size, the last elements deleted from the array are lost.

## Example 1

This example creates a process array of 100 *Picture* elements:

```
ARRAY PICTURE(agValues;100)
```

## Example 2

This example creates a local array of 100 rows of 50 *Picture* elements:

```
ARRAY PICTURE($agValues;100;50)
```

## Example 3

This example creates an interprocess array of *Picture* elements and loads each picture into one of the elements of the array. The array's size is equal to the number of *'PICT'* resources available to the database. The array's resource name starts with *"User Intf/"*:

```
RESOURCE LIST("PICT";$aiResIDs;$asResNames)
ARRAY PICTURE(◊agValues;Size of array($aiResIDs))
$vlPictElem:=0
For($vlElem;1;Size of array(◊agValues))
   If($asResNames{$vlElem}="User Intf/@")
      $vlPictElem:=$vlPictElem+1
      GET PICTURE RESOURCE("PICT";$aiResIDs{$vlElem};$vgPicture)
      ◊agValues{$vlPictElem}:=$vgPicture
   End if
End for
ARRAY PICTURE(◊agValues;$vlPictElem)
```

# ARRAY POINTER

ARRAY POINTER ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ➡ | Name of the array |
| size | Longint | ➡ | Number of elements in the array, or Number of rows if size2 is specified |
| size2 | Longint | ➡ | Number of columns in a two-dimensional array |

## Description

The **ARRAY POINTER** command creates or resizes an array of *Pointer* elements in memory.

- The *arrayName* parameter is the name of the array.
- The *size* parameter is the number of elements in the array.
- The *size2* parameter is optional; if *size2* is specified, the command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the firt dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying **ARRAY POINTER** to an existing array:

- If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to null *pointer*. This means that **Nil** applied to one of these elements will return True.
- If you reduce the array size, the last elements deleted from the array are lost.

## Example 1

This example creates a process array of 100 *Pointer* elements:

```
ARRAY POINTER(apValues;100)
```

## Example 2

This example creates a local array of 100 rows of 50 *Pointer* elements:

```
ARRAY POINTER($apValues;100;50)
```

## Example 3

This example creates an interprocess array of *Pointer* elements and sets each element pointing to the table whose number is the same as the element. The size of the array is equal to the number of tables in the database. In the case of a deleted table, the row will return **Nil**.

```
ARRAY POINTER(◊apValues;Get last table number)
For($vlElem;1;Size of array(◊apValues);1;-1)
   If(Is table number valid($vlElem))
      ◊apValues{$vlElem}:=Table($vlElem)
   End if
End for
```

# ARRAY REAL

ARRAY REAL ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ⇒ | Name of the array |
| size | Longint | ⇒ | Number of elements in the array or Number of rows if size2 is specified |
| size2 | Longint | ⇒ | Number of columns in a two-dimensional array |

## Description

The **ARRAY REAL** command creates and/or resizes an array of *Real* elements in memory.

- The arrayName parameter is the name of the array.
- The *size* parameter is the number of elements in the array.
- The *size2* parameter is optional; if *size2* is specified, the command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying **ARRAY REAL** to an existing array:

- If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to 0.
- If you reduce the array size, the last elements deleted from the array are lost.

## Example 1

This example creates a process array of 100 Real elements:

```
ARRAY REAL(arValues;100)
```

## Example 2

This example creates a local array of 100 rows of 50 Real elements:

```
ARRAY REAL($arValues;100;50)
```

## Example 3

This example creates an interprocess array of 50 Real elements and sets each element to its element number:

```
ARRAY REAL(◊arValues;50)
For($vlElem;1;50)
   ◊arValues{$vlElem}:=$vlElem
End for
```

# ARRAY TEXT

ARRAY TEXT ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ➡ | Name of the array |
| size | Longint | ➡ | Number of elements in the array or Number of rows if size2 is specified |
| size2 | Longint | ➡ | Number of columns in a two-dimensional array |

## Description

The **ARRAY TEXT** command creates and/or resizes an array of *Text* elements in memory.

- The *arrayName* parameter is the name of the array.
- The *size* parameter is the number of elements in the array.
- The *size2* parameter is optional; if *size2* is specified, the command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying **ARRAY TEXT** to an existing array:

- If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to "" (empty string).
- If you reduce the array size, the last elements deleted from the array are lost.

## Example 1

This example creates a process array of 100 *Text* elements:

```
ARRAY TEXT(atValues;100)
```

## Example 2

This example creates a local array of 100 rows of 50 *Text* elements:

```
ARRAY TEXT($atValues;100;50)
```

## Example 3

This example creates an interprocess array of 50 *Text* elements and sets each element to the value “Element #” followed by its element number:

```
ARRAY TEXT(◊atValues;50)
For($vlElem;1;50)
   ◊atValues{$vlElem}:="Element #"+String($vlElem)
End for
```

# ARRAY TIME

ARRAY TIME ( arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrayName | Array | ⇒ | Name of array |
| size | Longint | ⇒ | Number of array elements or Number of arrays if size2 is specified |
| size2 | Longint | ⇒ | Number of 2D array elements |

## Description

The **ARRAY TIME** command creates and/or resizes an array of Time type elements in memory.

**Reminder:** In 4D, times can be processed as numeric values. In 4D versions prior to v14, you had to combine a longint array with a display format in order to manage an array of times.

The *arrayName* parameter is the name of the array.

The *size* parameter is the number of array elements.

The *size2* parameter is optional. If you pass it, this command creates a two-dimensional array. In this case, *size* specifies the number of rows and *size2* the number of columns in each array. Each row in a two-dimensional array can be processed both as an element and an array. This means that when you work with the first dimension of a two-dimensional array, you can insert and remove entire arrays using other commands in this theme.

When you apply the **ARRAY TIME** command to an existing array:

- If you enlarge its size, existing elements are not changed and new elements are initialized to the null time value (00:00:00).
- If you reduce its size, elements at the "bottom" of the array are deleted and lost.

When you apply **SELECTION TO ARRAY** or **SELECTION RANGE TO ARRAY** to a Time type field, note that they only create a Time type array if the array has not already been defined as another type, such as Longint for example.

## Example 1

This example creates a process array containing 100 Time-type elements:

```
ARRAY TIME(arrTimes;100)
```

## Example 2

This example creates a local array of 100 rows each containing 50 Time-type elements:

```
ARRAY TIME($arrTimes;100;50)
```

## Example 3

Since time arrays accept numeric values, the following code is valid:

```
ARRAY TIME($arrTimeValues;10)
$CurTime:=Current time+1
APPEND TO ARRAY($arrTimeValues;$CurTime)
$Found:=Find in array($arrTimeValues;$CurTime)
```

# ARRAY TO LIST

ARRAY TO LIST ( array ; list {; itemRefs} )

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ➡ | Array from which to copy array elements |
| list | String, ListRef | ➡ | Name or reference of list into which to copy array elements |
| itemRefs | Array | ➡ | Numeric array of item reference numbers |

## Description

The **ARRAY TO LIST** command creates or replaces the hierarchical list or the choice list (created in the List editor) that is specified in *list* using the elements of the array *array.*

In the *list* parameter, you can either pass a choice list (string) or a hierarchical list reference (*ListRef*). In the latter case, this list must have already been created previously (for example using the **New list** command) in order for this command to work.

The optional *itemRefs* parameter, if specified, must be a numeric array synchronized with the array *array*. Each element, then, indicates the list item reference number for the corresponding element in *array*. If you omit this parameter, 4D automatically sets the list item reference numbers to 1, 2... N.

**Compatibility Note:** The **ARRAY TO LIST** command must be used with caution because of the following limitations:

- This command only lets you set first-level items of the list.
- When you use this command with a choice list, it modifies the structure of the application (lists are saved in the structure file), so any modifications made locally are lost when the structure file is updated in production.
- This command cannot be used in a component with a choice list because they are loaded with their structure as read only.

You can use **ARRAY TO LIST** to build a list based on the elements of an array. However, to free yourself from these restrictions and make full use of the lists of values, we recommend using the commands of the **Hierarchical Lists** theme.

## Example

The following example copies the array *atRegions* to the list called “Regions:”

```
ARRAY TO LIST(atRegions;"Regions")
```

## Example

You want to put the distinct values of a field into a list, for example to create a hierarchical pop-up menu. You can write:

```
ALL RECORDS([Company])
DISTINCT VALUES([Company]country;$arrCountries)
CountryList:=New list
ARRAY TO LIST($arrCountries;CountryList)
```

## Error management

An error *-9957* is generated when **ARRAY TO LIST** is applied to a list that is currently being edited in the Design environment List Editor. You can catch this error using an **ON ERR CALL** project method.

# ARRAY TO SELECTION

ARRAY TO SELECTION {( array ; aField {; array2 ; aField2 ; ... ; arrayN ; aFieldN}{; *} )}

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ⇒ | Array to copy to the selection |
| aField | Field | ⇐ | Field to receive the array data |
| * | Operator | ⇒ | Await execution |

## Description

The **ARRAY TO SELECTION** command copies one or more arrays into a selection of records. All fields listed must belong to the same table.

If a selection exists at the time of the call, the elements of the array are put into the records, based on the order of the array and the order of the records. If there are more elements than records, new records are created. The records, whether new or existing, are automatically saved.

**Note:** Since it can create new records, this command does not take a table's read-only state (if any) into account (see **Record Locking**).

All the arrays must have the same number of elements. If the arrays are of different sizes, a syntax error is generated.

This command does the reverse of **SELECTION TO ARRAY**. However, the **ARRAY TO SELECTION** command does not allow fields from different tables, including related tables, even when an automatic relation exists.

When you pass the * parameter, 4D does not execute the corresponding statement line immediately but instead stores it in memory; this way you can stack several lines ending with an *. All of these lines awaiting execution are executed by one final **ARRAY TO SELECTION** statement that does not have the * parameter. For this reason, the command can now be called without any parameters.
As with the **QUERY** command, this lets you break up a complex statement into a set of lines, which is easier to read and to maintain. You can also insert intermediary statements.

**WARNING:** Use **ARRAY TO SELECTION** with caution, because it overwrites information in existing records. If a record is locked by another process during the execution of **ARRAY TO SELECTION**, that record is not modified. Any locked records are put into the process set called *LockedSet*. After **ARRAY TO SELECTION** has executed, you can test the set *LockedSet* to see if any records were locked.

**Note**: This command does not take into account the read-only/read-write state of the table containing the field.

**4D Server:** The command is optimized for 4D Server. Arrays are sent by the client machine to the server, and the records are modified or created on the server machine. As such a request is handled synchronously, the client machine must wait for the operation to be completed successfully. In the multi-user or multi-process environment, any records that are locked will not be overwritten.

## Example 1

In the following example, the two arrays *asLastNames* and *asCompanies* place data in the *[People]* table. The values from the array *asLastNames* area placed in the field *[People]Last Name* and the values from the array *asCompanies* are placed in the field *[People]Company*:

```
ARRAY TO SELECTION(asLastNames;[People]Last Name;asCompanies;[People]Company)
```

## Example 2

This example duplicates a selection of records:

```
ARRAY TEXT(a1;0)
ARRAY TEXT(a2;0)
ARRAY TEXT(a3;0)
ARRAY TEXT(a4;0)

ALL RECORDS([Table_1])
For($i;1;Get last field number(1))
```

```
   $p:=Get pointer("a"+String($i))
   SELECTION TO ARRAY(Field(1;$i)->;$p->;*)
  // Deferred building of arrays
End for
SELECTION TO ARRAY //Execution of statements

REDUCE SELECTION([Table_1];0)
For($i;1;Get last field number(1))
   $p:=Get pointer("a"+String($i))
   ARRAY TO SELECTION($p->;Field(1;$i)->;*)
  // Building selection
End for
ARRAY TO SELECTION //Execution of statements
```

# BOOLEAN ARRAY FROM SET

BOOLEAN ARRAY FROM SET ( booleanArr {; set} )

| Parameter | Type | | Description |
|---|---|---|---|
| booleanArr | Boolean array | ⇦ | Array to indicate if a record is in the set or not |
| set | String | ⇨ | Name of the set or UserSet if this parameter is omitted |

## Description

The **BOOLEAN ARRAY FROM SET** command fills an array of Booleans indicating if each record in the table is or is not in *set*.

The elements in the array are ordered in the order in which the records are created in the table (absolute record numbers). If N is the number of records in the table, element 0 of the array corresponds to record number 0, element 1 of the array corresponds to record number 1, etc.

Each element of the array is:

- **True** if the corresponding record belongs to the set.
- **False** if the corresponding record does not belong to the set.

**Warning:** The total number of elements in the *booleanArr* array is not significant. For structural reasons, this number can be different from the number of records actually present in the table. Possible extra elements are set to **False**.

If you don't pass the *set* parameter, the command will use *UserSet* in the current process.

# COPY ARRAY

COPY ARRAY ( source ; destination )

| Parameter | Type | | Description |
|---|---|---|---|
| source | Array | ⇨ | Array from which to copy |
| destination | Array | ⇦ | Array to which to copy |

## Description

The **COPY ARRAY** command creates or overwrites the destination array *destination* with the exact contents, size, and type of the source array *source*.

The *source* and *destination* arrays can be local, process, or interprocess arrays. When copying arrays, the scope of the array does not matter.

**Note:** In compiled mode, the *destination* array must be of the same type as the *source* array.

## Example

The following example fills the array named C. It then creates a new array, named D, of the same size as C and with the same contents:

```
ALL RECORDS([People]) ` Select all records in People
SELECTION TO ARRAY([People]Company;C) ` Move company field data into array C
COPY ARRAY(C;D) ` Copy the array C to the array D
```

# Count in array

Count in array ( array ; value ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ⇒ | Array where count should occur |
| value | Expression | ⇒ | Value to count |
| Function result | Longint | ⇐ | Number of instances found |

## Description

The **Count in array** command returns the number of times *value* is found in *array*.

This command can be used with the following array types: Text, Alpha, number, Date, Pointer and Boolean. The *array* and *value* parameters must be the same type or compatible.

If no element in *array* matches *value*, the command returns 0.

## Example

The following example allows displaying the number of selected lines in a list box:

```
 //tBList is the name of a List box column array
ALERT(String(Count in array(tBList;True))+" line(s) selected in the list box")
```

# DELETE FROM ARRAY

DELETE FROM ARRAY ( array ; where {; howMany} )

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ⇒ | Array from which to delete elements |
| where | Longint | ⇒ | Element at which to begin deletion |
| howMany | Longint | ⇒ | Number of elements to delete, or 1 element if omitted |

## Description

The **DELETE FROM ARRAY** command deletes one or more elements from *array*. Elements are deleted starting at the element specified by *where*.

The *howMany* parameter is the number of elements to delete. If *howMany* is not specified, then one element is deleted. The size of the array shrinks by *howMany*.

## Example 1

The following example deletes three elements, starting at element 5:

```
DELETE FROM ARRAY(anArray;5;3)
```

## Example 2

The following example deletes the last element from an array, if it exists:

```
$vlElem:=Size of array(anArray)
If($vlElem>0)
   DELETE FROM ARRAY(anArray;$vlElem)
End if
```

# DISTINCT ATTRIBUTE PATHS

DISTINCT ATTRIBUTE PATHS ( objectField ; pathArray )

| Parameter | Type | | Description |
|---|---|---|---|
| objectField | Field | ⇒ | Indexed object field |
| pathArray | Text array | ⇐ | Array to receive list of distinct paths |

## Description

The **DISTINCT ATTRIBUTE PATHS** command returns the list of distinct paths found in the indexed object field you passed in *objectField* for the current selection of the table to which this field belongs.

In *objectField*, you must pass an Object type field that is indexed; otherwise, an error is returned.

After the call, the size of *pathArray* is equal to the number of distinct paths found in the selection. Paths to nested object attributes are returned using the standard dot notation, for example "company.address.number". Keep in mind that object attribute names are case-sensitive. The command does not change the current selection or the current record.

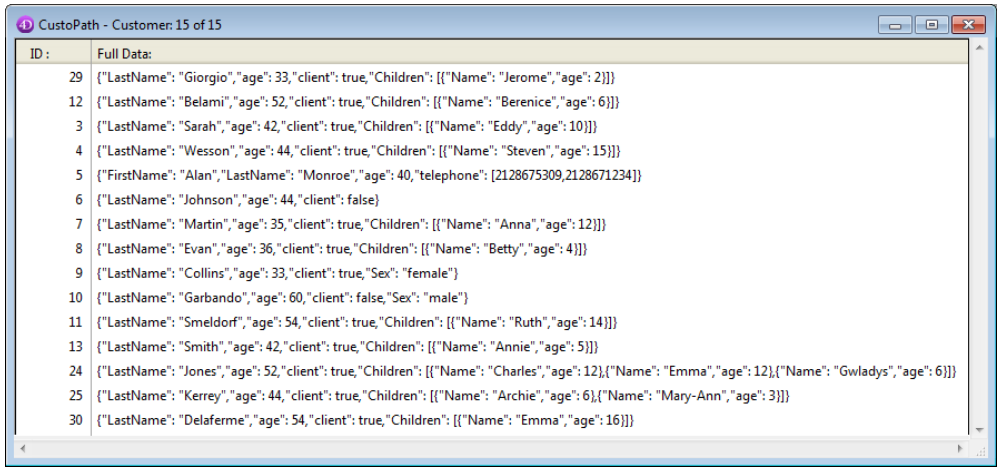
In *pathArray*, the list of distinct paths is returned in alphabetical (diacritic) order.

**Notes:**

- Records with an undefined value in the *objectField* are not taken into account.
- Attribute paths created during a transaction are taken into account by the command. It is important to note that these paths are kept in the index of the object field even if the transation has been canceled.

## Example

Your database contains a [Customer]full_Data (indexed) object field with 15 records:




If you execute this code:

```
ARRAY TEXT(aTPaths;0)
ALL RECORDS([Customer])
DISTINCT ATTRIBUTE PATHS([Customer]full_Data;aTPaths)
```

The *aTPaths* array receives the following elements:

| Element | Value |
| --- | --- |
| 1 | "age" |
| 2 | "Children" |
| 3 | "Children[]" |
| 4 | "Children[].age" |
| 5 | "Children[].Name" |
| 6 | "Children.length" |
| 7 | "client" |
| 8 | "FirstName" |
| 9 | "LastName" |
| 10 | "Sex" |
| 11 | "telephone" |
| 12 | "telephone[]" |
| 13 | "telephone.length" |

**Note:** "length" is a *virtual property* that is automatically available for all array type attributes. It provides the size of the array, i.e. the number of elements, and can be used in queries. For more information, please refer to the **Using the .length virtual property** paragraph.

# DISTINCT ATTRIBUTE VALUES

DISTINCT ATTRIBUTE VALUES ( objectField ; path ; valuesArray )

| Parameter | Type | | Description |
|---|---|---|---|
| objectField | Field | ➡ | Object field from which to get the list of distinct attribute values |
| path | Text | ➡ | Path of attribute whose distinct values you want to get |
| valuesArray | Text array, Longint array, Boolean array, Date array, Time array | ⬅ | Distinct values in attribute path |

## Description

The **DISTINCT ATTRIBUTE VALUES** command creates and populates the *valuesArray* with non-repeated (unique) values coming from the *path* attribute in the *objectField* field for the current selection of the table to which this field belongs. Note that *objectField* must be of the Object type, otherwise an error is returned. The command can be used with indexed or non-indexed fields.

Pass a valid attribute path in *path*. Use the standard dot notation to define paths to nested attributes, for example "company.address.number". Keep in mind that object attribute names are case-sensitive.

The array you passed in *valuesArray* must be of the same type as the values stored in the attribute *path*. Values must be scalar and can be of the Text, number, Boolean, Date, or Time type (pointers, objects, BLOBs or images are not supported). Make sure that all field attribute values are of the same type; otherwise, an error is returned. For example, if the *path* attribute contains "Monday" in one record and 10125 in another record, an error will be returned.

If the command is called from within a transaction, records created during the transaction are taken into account.

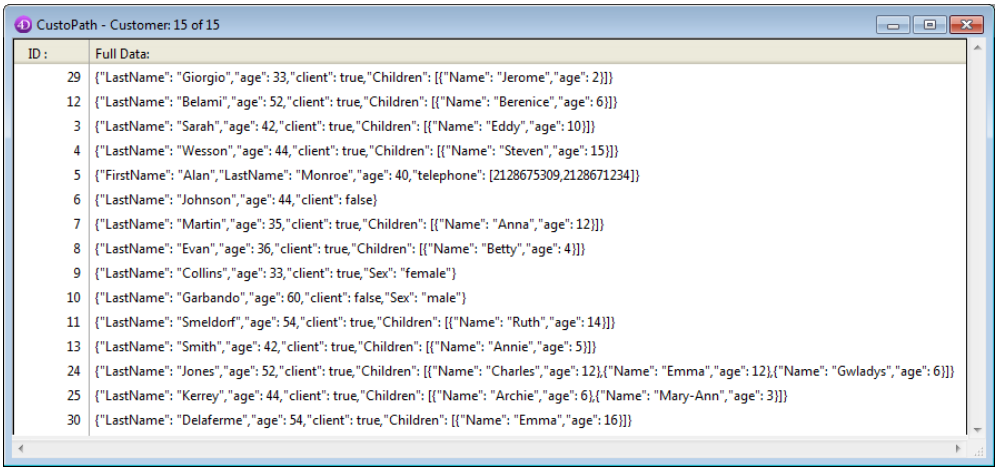
After the call, the size of the array is equal to the number of distinct values found in the selection. The command does not change the current selection or the current record.

### Using the .length virtual property

You can use the "length" virtual property with this command. It is automatically available for all attributes of the array type, and provides the size of the array, i.e. the number of elements it contains. This property is designed to be used in queries (see **QUERY BY ATTRIBUTE**). You can also use it with the **DISTINCT ATTRIBUTE VALUES** command to get the different array sizes for an attribute.

## Example

Your database contains a [Customer]full_Data object field with 15 records:




If you execute this code:

```
ARRAY LONGINT(aLAges;0)
ARRAY LONGINT(aLAgesChild;0)
ARRAY LONGINT(aLChildNum;0)
ALL RECORDS([Customer])
 //get the distinct values for the "age" attribute
DISTINCT ATTRIBUTE VALUES([Customer]full_Data;"age";aLAges)
```

```
 //get the distinct values for the "age" attribute within the "Children" array
DISTINCT ATTRIBUTE VALUES([Customer]full_Data;"Children[].age";aLAgesChild)
 //get the distinct numbers of children by using the length virtual property
DISTINCT ATTRIBUTE VALUES([Customer]full_Data;"Children.length";aLChildNum)
```

The *aLAges* array receives the following elements:

| Element | Value |
|---|---|
| 1 | 33 |
| 2 | 35 |
| 3 | 36 |
| 4 | 40 |
| 5 | 42 |
| 6 | 44 |
| 7 | 52 |
| 8 | 54 |
| 9 | 60 |

The *aLAgesChild* array receives the following elements:

| Element | Value |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 5 |
| 5 | 6 |
| 6 | 10 |
| 7 | 12 |
| 8 | 14 |
| 9 | 15 |
| 10 | 16 |

The *aLChildNum* array receives the following elements:

| Element | Value |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |

# DISTINCT VALUES

DISTINCT VALUES ( aField ; array {; countArray} )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ➡ | Indexable field to use for data |
| array | Array | ⬅ | Array to receive field data |
| countArray | Longint array, Real array | ⬅ | Array to receive count of each value |

## Description

The **DISTINCT VALUES** command creates and populates the array *array* with non-repeated (unique) values coming from the field *aField* for the current selection of the table to which the field belongs and, optionally, returns the number of occurrences of each value in the *countArray* parameter.

You can pass to **DISTINCT VALUES** any **indexable** field, that is, whose type supports indexing without necessarily being indexed.
However, executing this command on unindexed fields will be slower. Also note that, in this case, the command loses the current record.

**DISTINCT VALUES** browses and retains the non-repeated values present only in the currently selected records.

**Note:** When the **DISTINCT VALUES** command is called during a transaction (that has not yet finished), it **will take** into account records created during that transaction.

The array used by **DISTINCT VALUES** must be of the same type as the field passed as first parameter, otherwise the array is retyped. There is one exception to this rule: if the field is of the Picture type (and is associated with a keyword index), the corresponding array must be of the Text type.

After the call, the size of the array is equal to the number of distinct values found in the selection. The command does not change the current selection or the current record. The **DISTINCT VALUES** command uses the index of the field, so the elements in *array* are returned sorted in ascending order. If this is the order you need, you do not need to call **SORT ARRAY** after using **DISTINCT VALUES**.

**Note:** When **DISTINCT VALUES** is executed with a text or picture field associated with a keyword index, the command fills the array with the keywords of the index. Unlike other types of data, the values returned differ according to the existence of the index. In the case of a Text field, the keyword index is always taken into account, even when the field is also associated with a standard index. If the Text or Picture field is not associated with a keyword index, the array is returned empty.

The command accepts a *countArray* array as an optional parameter. When it is passed, this array returns, for each non-repeated value in *aField*, the number of occurrences detected in the current selection. The *countArray* array is automatically sized to the number of elements in *array*. For example, for a selection that contains three records with field values "A", "B", and "A", *array* will contain {A;B} and *countArray* will contain {2;1}. You can pass either a Longint array or a Real array in *countArray*.

**Note:** The *countArray* parameter is not supported for text or picture fields that are associated with keyword indexes (in this context, it is returned empty).

**WARNING: DISTINCT VALUES** can create large arrays depending on the size of the selection and the number of different values in the records. Arrays reside in memory, therefore it is a good idea to test the result after the completion of the command. To do so, test the size of the resulting array or cover the call to the command, using an **ON ERR CALL** project method.

**4D Server:** The command is optimized for 4D Server. The array is created and the values are calculated on the server machine; the array is then sent, in its entirety, to the client.

**Note:** This command does not support Object type fields.

## Example 1

The following example creates a list of cities from the current selection and tells the user the number of cities in which the firm has stores:

```
ALL RECORDS([Retail Outlets]) ` Create a selection of records
DISTINCT VALUES([Retail Outlets]City;asCities)
```

```
ALERT("The firm has stores in "+String(Size of array(asCities))+" cities.")
```

## Example 2

You want to get a complete list of keywords contained in the keyword index for the "Pictures" field:

```
ALL RECORDS([PICTURES])
ARRAY TEXT(<>_MyKeywords;10)
DISTINCT VALUES([PICTURES]Photos;<>_MyKeywords)
```

## Example 3

To compute statistics, you want to sort the number of distinct values in a field in descending order:

```
ARRAY TEXT($_issue_type;0)
ARRAY LONGINT($_issue_type_instance;0)
DISTINCT VALUES([Issue]iType;$_issue_type;$_issue_type_instances)
SORT ARRAY($_issue_type_instances;$_issue_type;<)
```

# Find in array

Find in array ( array ; value {; start} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ➡ | Array to search |
| value | Expression | ➡ | Value of same type to search in the array |
| start | Longint | ➡ | Element at which to start searching |
| Function result | Longint | ↩ | Number of the first element in array that matches value |

## Description

The **Find in array** command returns the number of the first element in *array* that matches *value*.

**Find in array** can be used with Text, String, Numeric, Date, Pointer, and Boolean arrays. The *array* and *value* parameters must be of the same type.

If no match is found, **Find in array** returns -1.

If *start* is specified, the command starts searching at the element number specified by *start*. If *start* is not specified, the command starts searching at element 1.

## Example 1

The following project method deletes all empty elements from the string or text array whose pointer is passed as parameter:

```
  ` CLEAN UP ARRAY project method
  ` CLEAN UP ARRAY ( Pointer )
  ` CLEAN UP ARRAY ( -> Text or String array )

C_POINTER($1)
Repeat
   $vlElem:=Find in array($1->;"")
   If($vlElem>0)
      DELETE FROM ARRAY($1->;$vlElem)
   End if
Until($vlElem<0)
```

After this project method is implemented in a database, you can write:

```
ARRAY TEXT(atSomeValues;...)
  ` ...
  ` Do plenty of things with the array
  ` ...
  ` Eliminate empty string elements
CLEAN UP ARRAY(->atSomeValues)
```

## Example 2

The following project method selects the first element of an array whose pointer is passed as the first parameter that matches the value of the variable or field whose pointer is passed as parameter:

```
  ` SELECT ELEMENT project method
  ` SELECT ELEMENT ( Pointer ; Pointer)
  ` SELECT ELEMENT ( -> Text or String array ; -> Text or String variable or field )

$1->:=Find in array($1->;$2->)
If($1->=-1)
   $1->:=0 ` If no element was found, set the array to no selected element
End if
```

After this project method is implemented in a database, you can write:

```
   ` asGender pop-up menu object method
Case of
   :(Form event=On Load)
      SELECT ELEMENT(->asGender;->[People]Gender)

End case
```

**Note:** This example uses the **selected element** of the array. Keep in mind that the selected element is not meaningful if the array contains more than 32,767 elements (see **Arrays and Form Objects**). In this case, you need to use a longint variable to store the result of **Find in array**.

# Find in sorted array

Find in sorted array ( array ; value ; > or < {; posFirst {; posLast}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ⇒ | Array to search |
| value | Expression | ⇒ | Value (same type as array) to search for in the array |
| > or < | Operator | ⇒ | > if array is sorted in ascending order, < if it is sorted in descending order |
| posFirst | Longint | ⇐ | Position of its first occurrence if the value is found; otherwise position where the value should be inserted |
| posLast | Longint | ⇐ | Position of its last occurrence if the value is found; otherwise same as posFirst |
| Function result | Boolean | ⇐ | True if at least one element in array matches the value, False otherwise |

## Description

The **Find in sorted array** command returns **true** if at least one element in the sorted *array* matches the *value*, and optionally returns position(s) of matched element(s). Unlike **Find in array**, **Find in sorted array** only works with a sorted *array* and provides information about the position of occurrences, which allows you to insert elements if necessary.

The *array* must be already sorted and must match the ordering specified by the *> or <* parameter (i.e. the "greater than" symbol for ascending order and the "lower than" symbol for descending order). The **Find in sorted array** command will take advantage of the sort and use a *binary search* algorithm, which is much more efficient for large arrays (for more information, please refer to the binary search algorithm page on Wikipedia). However, if the array is not properly sorted, the result may be incorrect.

The command will ignore the sort indication and behave like a standard **Find in array** (sequential search, returning -1 for *posFirst* and *posLast* if the *value* is not found) in any of the following cases:

- if the array type cannot be sorted (e.g. pointer arrays),
- if the array is of type boolean (not accurate),
- if the database is not Unicode (compatibility mode) and the array is a string or text array,
- when searching in a text array for a string that includes a wildcard ('@') at the beginning or in the middle of the string (using a binary search with such a wildcard character is not possible because matching elements may be non-contiguous in the array).

In case the command returns **False**, the value returned in *posFirst* can be passed to **INSERT IN ARRAY** to insert the *value* into the array while keeping the array sorted. This sequence is faster than inserting a new item at the end of the array and then calling **SORT ARRAY** to move it to the right place.

The value returned in *posLast* can be combined with the value returned in *posFirst* to iterate on each element of the array matching the *value* (with a **For...End for** loop) or to find the number of occurrences (as would be found by **Count in array**, but faster).

## Example 1

You want to insert a value, if necessary, while keeping the array sorted:

```
C_LONGINT($pos)
If(Find in sorted array($array ;$value ;>;$pos)
   ALERT("Found at pos "+String($pos))
Else
   INSERT IN ARRAY($array ;$pos)
   $array{$pos}:=$value
End if
```

## Example 2

You want to find the number of occurrences of strings starting with "test" and create a string that concatenates all these elements:

```
C_LONGINT($posFirst ;$posLast)
C_TEXT($output)
```

```
If(Find in sorted array($array ;"test@";>;$posFirst ;$posLast))
   $output:="Found "+String($posLast-$posFirst+1)+" results :¥n"
End if
For($i ;$posFirst ;$posLast)
   $output:=$output+$array{$i}+"¥n"
End for
```

# INSERT IN ARRAY

INSERT IN ARRAY ( array ; where {; howMany} )

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ⇒ | Name of the array |
| where | Longint | ⇒ | Where to insert the elements |
| howMany | Longint | ⇒ | Number of elements to be inserted, or 1 element if omitted |

## Description

The **INSERT IN ARRAY** command inserts one or more elements into the array *array*. The new elements are inserted before the element specified by *where*, and are initialized to the empty value for the array type. All elements beyond *where* are consequently moved within the array by an offset of one or the value you pass in *howMany*.

If *where* is greater than the size of the array, the elements are added to the end of the array.

The *howMany* parameter is the number of elements to insert. If *howMany* is not specified, then one element is inserted. The size of the array grows by *howMany*.

## Example 1

The following example inserts five new elements, starting at element 10:

```
INSERT IN ARRAY(anArray;10;5)
```

## Example 2

The following example appends an element to an array:

```
$vlElem:=Size of array(anArray)+1
INSERT IN ARRAY(anArray;$vlElem)
anArray{$vlElem}:=...
```

# LIST TO ARRAY

LIST TO ARRAY ( list ; array {; itemRefs} )

| Parameter | Type | | Description |
|---|---|---|---|
| list | String, ListRef | ⇒ | Name or Reference of list from which to copy the first level items |
| array | Array | ⇐ | Array to which to copy the list items |
| itemRefs | Array | ⇐ | List item reference numbers |

## Description

The **LIST TO ARRAY** command creates or overrides the array *array* with the first level items of the list or choice list designated by *list*.

In the list parameter, you can pass either the name of a choice list (string), or a reference to a hierarchical list (*ListRef*).

If you do not set the array as an Alpha or Text type beforehand, **LIST TO ARRAY** creates a new Text array by default.

**Note:** In compiled mode, the *array* must have been defined previously and cannot be retyped.

The optional *itemRefs* parameter (a numeric array) returns the list item reference numbers.

You can use **LIST TO ARRAY** to build an array based on the first level items of a list. However, this command does not allow you to work with any of the list's child items. When working with hierarchical lists, we recommend that you use the hierarchical lists commands, in particular **Load list**.

## Example 1

The following example copies the items of a list called Regions into an array called *atRegions*:

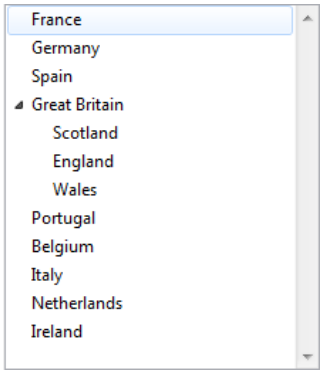
```
LIST TO ARRAY("Regions";atRegions)
```

## Example 2

Given a hierarchical list created as follows:

```
myList2:=New list
APPEND TO LIST(myList2;"Scotland";1)
APPEND TO LIST(myList2;"England";2)
APPEND TO LIST(myList2;"Wales";3)
myList1:=New list
APPEND TO LIST(myList1;"France";1)
APPEND TO LIST(myList1;"Germany";2)
APPEND TO LIST(myList1;"Spain";3)
APPEND TO LIST(myList1;"Great Britain";4;myList2;True)
APPEND TO LIST(myList1;"Portugal";5)
APPEND TO LIST(myList1;"Belgium";6)
APPEND TO LIST(myList1;"Italy";7)
APPEND TO LIST(myList1;"Netherlands";8)
APPEND TO LIST(myList1;"Ireland";9)
```

This list can be represented as:




If you execute the following statement:

```
LIST TO ARRAY(myList1;$MyArray)
```

...you get

```
$MyArray{1}="France"
$MyArray{2}="Germany"
$MyArray{3}="Spain"
$MyArray{4}="Great Britain"
$MyArray{5}="Portugal"
...
```

# LONGINT ARRAY FROM SELECTION

LONGINT ARRAY FROM SELECTION ( aTable ; recordArray {; selection} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table of the current selection |
| recordArray | Longint array | ⇐ | Array of record numbers |
| selection | String | ⇒ | Name of the named selection or the current selection if this parameter is omitted |

## Description

The **LONGINT ARRAY FROM SELECTION** command fills the *recordArray* array with the (absolute) record numbers that are in *selection*.

If you do not pass the *selection* parameter, the command will use the current selection of *aTable*.

**Note:** The array element number 0 is initialized to -1.

## Example

You want to retrieve the numbers of the records in the current selection:

```
ARRAY LONGINT($_arrRecNum;0) //mandatory for compiled mode
LONGINT ARRAY FROM SELECTION([Clients];$_arrRecNum)
```

# MULTI SORT ARRAY

MULTI SORT ARRAY ( array {; sort}{; array2 ; sort2 ; ... ; arrayN ; sortN} )

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ⇒ | Array(s) to be sorted |
| sort | Operator | ⇒ | ">" to sort by increasing order or "<" to sort by decreasing order If omitted no sort |

MULTI SORT ARRAY ( ptrArrayName ; sortArrayName )

| Parameter | Type | | Description |
|---|---|---|---|
| ptrArrayName | Pointer array | ⇒ | Array of array pointers |
| sortArrayName | Longint array | ⇒ | Sort order array (1 = sort by increasing order, -1 = sort by decreasing order, 0 = synchronization with previous sorts) |

## Description

The **MULTI SORT ARRAY** command enables you to carry out a multi-level sort on a set of arrays.

This command accepts two different syntaxes.

- **First syntax: MULTI SORT ARRAY (array{; sort}{; array2; sort2; ...; arrayN; sortN})**

This syntax is the simplest; it lets you directly pass the names of the synchronized arrays where you want to apply a multi-criteria sort.

You can pass an unlimited number of pairs (*array;> or <*) and/or only arrays. All the arrays passed as parameters are sorted in a synchronized manner.

You can pass arrays of any type except for Pointer or Picture arrays. You can sort an element of a two-dimensional array (i.e. *a2DArray{$vlThisElement}*), but you cannot sort the 2D array itself (i.e. *a2DArray*).

To use the contents of an array as sort criteria, pass the *sort* parameter. The value of the parameter (> or <) determines the order (ascending or descending) in which the array will be sorted. If the *sort* parameter is omitted, the contents of the array are not used as sort criteria.

**Note:** Keep in mind that at least one sort criterion must be passed in order for the command to work. If no sort criterion is set, an error is generated.

The sort levels are determined by the order in which the arrays are passed to the command: the position of an array with a sort criterion in the syntax determines its sort level.

- **Second syntax: MULTI SORT ARRAY (ptrArrayName; sortArrayName)**

This syntax, more complex, is also invaluable for generic developments (for example, you can create a generic method for sorting arrays of all types, or yet again, create the equivalent of a generic **SORT ARRAY** command).

The *ptrArrayName* parameter contains the name of an array of array pointers; each element of this array is a pointer designating an array to be sorted. The sorts are performed in the order of the array pointers defined by *ptrArrayName*.
**Warning:** all the arrays pointed to by *ptrArrayName* must have the same number of elements.

**Note:** *ptrArrayName* can be an array of local ($ptrArrayName), process (ptrArrayName) or inter-process (<>ptrArrayName) pointers. Conversely, the elements of this array must point to process or inter-process arrays only.

The *sortArrayName* parameter contains the name of an array in which each element indicates the sorting order (-1, 0 or 1) of the element of the corresponding array of pointers:
-1 = Sort by decreasing order.
0 = The array is not used as a sorting criterion but must be sorted according to the other sorts.
1 = Sort by increasing order.

**Note:** You cannot sort arrays of the Pointer or Picture type. You can sort an element of a two-dimensional array (i.e. a2DArray{$vlThisElement}), but you cannot sort the 2D array itself (i.e. a2DArray).

For each element of the *ptrArrayName* array, there must be a corresponding element of the *sortArrayName* array. Both arrays must therefore have exactly the same number of elements.

## Example 1

The following example uses the first syntax: it creates four arrays and sorts them by city (ascending order) then by salary (descending order) with the last two arrays, *names_array* and *telNum_array*, being synchronized according to the previous sort criteria:

```
ALL RECORDS([Employees])
SELECTION TO ARRAY([Employees]City;cities;[Employees]Salary;salaries;[Employees]Name;
names;[Employees]TelNum;telNums)
MULTI SORT ARRAY(cities;>;salaries;<;names;telNums)
```

If you want for the names array to be used as the third sort criteria, just add > or < after the *names_array* parameter. Note that the syntax:

```
MULTI SORT ARRAY(cities;>;salaries;names;telNums)
```

is equivalent to:

```
SORT ARRAY(cities;salaries;names;telNums;>)
```

## Example 2

The following example uses the second syntax: it creates four arrays and sorts them by city (increasing order) and company (decreasing order); the last two arrays, names_Array and telNum_Array, being synchronized according to previous sort criteria:

```
ALL RECORDS([Employees])
SELECTION TO ARRAY([Employees]City;cities;[Employees]Company;companies;[Employees]Name;
names;[Employees]TelNum;telNums)
ARRAY POINTER(pointers_Array;4)
ARRAY LONGINT(sorts_Array;4)
pointers_Array{1}:=->cities
sorts_Array{1}:=1
pointers_Array{2}:=->companies
sorts_Array{2}:=-1
pointers_Array{3}:=->names
sorts_Array{3}:=0
pointers_Array{4}:=->telNums
sorts_Array{4}:=0
MULTI SORT ARRAY(pointers_Array;sorts_Array)
```

If you want the array of names be used as a third sort criterion, you need to assign the value 1 to the sorts_Array{3} element. Or else, if you want the arrays to be sorted only by the city criterion, assign the value 0 to the sorts_Array{2}, sorts_Array{3} and sorts_Array{4} elements. In this way, you obtain an identical result to **SORT ARRAY***(cities;companies;names;telNums;>)*.

# SELECTION RANGE TO ARRAY

SELECTION RANGE TO ARRAY ( start ; end {; aField | aTable ; array} {; aField | aTable2 ; array2 ; ... ; aField | aTableN ; arrayN} )

| Parameter | Type | | Description |
|---|---|---|---|
| start | Longint | ➡ | Selected record number where data retrieval starts |
| end | Longint | ➡ | Selected record number where data retrieval ends |
| aField \| aTable | Field, Table | ➡ | Field to use for retrieving data or Table to use for retrieving record numbers |
| array | Array | ⬅ | Array to receive field data or record numbers |

## Description

**SELECTION RANGE TO ARRAY** creates one or more arrays and copies data from the fields or record numbers from the current selection into the arrays.

Unlike **SELECTION TO ARRAY**, which applies to the current selection in its entirety, **SELECTION RANGE TO ARRAY** only applies to the range of selected records specified by the parameters *start* and *end*.

The command expects you to pass in *start* and *end* the selected record numbers complying with the formula *1 <= start <= end <= Records in selection ([...])*.

If you pass *1 <= start = end < Records in selection ([...])*, you will load fields or get the record number from the record whose selected record is *start = end*.

If you pass incorrect selected record numbers, the command does the following:

- If *end > Records in selection ([...])*, it returns values from the selected record specified by *start* to the last selected record.
- If *start > end*, it returns values from the record whose selected record is *start* only.
- If both parameters are inconsistent with the size of the selection, it returns empty arrays.

Like **SELECTION TO ARRAY**, the **SELECTION RANGE TO ARRAY** command applies to the selection for the table specified in the first parameter.

Also like **SELECTION TO ARRAY**, **SELECTION RANGE TO ARRAY** can perform the following:

- Load values from one or several fields.
- Load Record numbers using the syntax *...;[table];Array;...*
- Load values from related fields, if there is a Many to One automatic relation between the tables or if you have previously called **SET AUTOMATIC RELATIONS** to change manual Many to One relations to automatic. In both cases, values can be loaded from tables through several levels of Many to One relations.

Each array is typed according to the field type.

When you apply **SELECTION RANGE TO ARRAY** to a Time type field, it is important to note that they only create a Time type array if the array has not already been defined as another type. For example, in the following context, the *myArray* array remains a Longint type array:

```
ARRAY LONGINT(myArray;0)
SELECTION TO ARRAY([myTable]myTimeField;myArray)
```

If you load record numbers, they are copied into a Long Integer array.

**Note:** You can call the **SELECTION RANGE TO ARRAY** command with just the *start* and *end* parameters. You use this special syntax to launch, on a limited selection, the execution of a deferred series of **SELECTION TO ARRAY** commands using the * parameter (see example 4).

**4D Server**: **SELECTION RANGE TO ARRAY** is optimized for 4D Server. Each array is created on the server and then sent, in its entirety, to the client machine.

**WARNING: SELECTION RANGE TO ARRAY** can create large arrays, depending on the range you specify in *start* and *end,* and on the type and size of the data you are loading. Arrays reside in memory, so it is a good idea to test the result after the command is completed. To do so, test the size of each resulting array or cover the call to the command, using an **ON ERR CALL** project method.

If the command is successful, the size of each resulting array is equal to *(end-start)+1*, except if the *end* parameter exceeded the number of records in the selection. In such a case, each resulting array contains *(Records in selection([...])-start)+1* elements.

## Example 1

The following code addresses the first 50 records from the current selection for the *[Invoices]* table. It loads the values from the *[Invoices]Invoice ID* field and the *[Customers]Customer ID* related field.

```
SELECTION RANGE TO ARRAY(1;50;[Invoices]Invoice ID;alInvoID;[Customers]Customer ID;alCustID)
```

## Example 2

The following code addresses the last 50 records from the current selection for the *[Invoices]* table. It loads the record numbers of the *[Invoices]* records as well as those of the *[Customers]* related records:

```
lSelSize:=Records in selection([Invoices])
SELECTION RANGE TO ARRAY(lSelSize-49;lSelSize;[Invoices];alInvRecN;[Customers];alCustRecN)
```

## Example 3

The following code process, in sequential “chunks”of 1000 records, a large selection that could not be downloaded in its entirety into arrays:

```
lMaxPage:=1000
lSelSize:=Records in selection([Phone Directory])
For($lPage ;1;1+((lSelSize-1)¥lMaxPage))
  ` Load the values and/or record numbers
   SELECTION RANGE TO ARRAY(1+(lMaxPage*($lPage-1));lMaxPage*$lPage;...;...;...;...;...;...)
  ` Do something with the arrays
End for
```

## Example 4

Use the first 50 current records of the [Invoices] table to load various arrays, in deferred execution:

```
 // Deferred statements
SELECTION TO ARRAY([Invoices]InvoiceRef;arrLInvRef;*)
SELECTION TO ARRAY([Invoices]Date;arrDInvDate;*)
SELECTION TO ARRAY([Clients]ClientRef;arrLClientRef;*)
 // Execution of deferred statements
SELECTION RANGE TO ARRAY(1;50)
```

# SELECTION TO ARRAY

SELECTION TO ARRAY {( aField | aTable ; array {; aField ; array {; aField2 ; array2 ; ... ; aFieldN ; arrayN}}{; *})}

| Parameter | Type | | Description |
|---|---|---|---|
| aField \| aTable | Field, Table | ➡ | Field to use for retrieving data or Table to use for retrieving record numbers |
| array | Array | ⬅ | Array to receive field data or record numbers |
| aField | Field | ➡ | Field to retrieve in array |
| array | Array | ⬅ | Array to receive field data |
| * | Operator | ➡ | Await execution |

## Description

The **SELECTION TO ARRAY** command creates one or more arrays and copies data in the fields or record numbers from the current selection into the arrays.

The command **SELECTION TO ARRAY** applies to the selection for the table designated by the first parameter (table name or field name). **SELECTION TO ARRAY** can perform the following:

- Load values from one or several fields.
- Load Record numbers from the table using the syntax *[table];array*
- Load values from related fields, provided that there is a Many to One automatic relation between the tables or provided that you have previously called **SET AUTOMATIC RELATIONS** to make manual Many to One relations automatic. In both cases, values are loaded from tables through several levels of Many to One relations.

Each array is typed according to the field type.

When you apply **SELECTION TO ARRAY** to a Time type field, it is important to note that they only create a Time type array if the array has not already been defined as another type. For example, in the following context, the *myArray* array remains a Longint type array:

```
ARRAY LONGINT(myArray;0)
SELECTION TO ARRAY([myTable]myTimeField;myArray)
```

If you load record numbers, they are copied into a Long Integer array.

When you pass the * parameter, 4D does not execute the corresponding statement line immediately but instead stores it in memory; this way you can stack several lines ending with an *. All of these lines awaiting execution are executed by one final **SELECTION TO ARRAY** statement that does not have the * parameter. For this reason, the command can now be called without any parameters. In this case, array types are verified when the final line (without the * parameter) is executed.
As with the **QUERY** command, this lets you break up a complex statement into a set of lines, which is easier to read and to maintain. You can also insert intermediary statements or build an array within a loop (see example 2 of the **ARRAY TO SELECTION** command).

**4D Server:** The **SELECTION TO ARRAY** command is optimized for 4D Server. Each array is created on the server and then sent, in its entirety, to the client machine.

**WARNING:** The **SELECTION TO ARRAY** command can create large arrays, depending on the size of the current selection and on the type and size of the data you are loading. Arrays reside in memory, so it is a good idea to test the result after the command is completed. To do so, test the size of each resulting array or cover the call to the command, using an **ON ERR CALL** project method.

**Note:** After a call to **SELECTION TO ARRAY**, the current selection and current record remain the same, but the current record is no longer loaded. If you need to use the values of the fields in the current record, use the **LOAD RECORD** command after the **SELECTION TO ARRAY** command.

## Example 1

In the following example, the *[People]* table has an automatic relation to the [Company] table. The two arrays *asLastName* and *asCompanyAddr* are sized according to the number of records selected in the *[People]* table and will contain information from both tables:

```
SELECTION TO ARRAY([People]Last Name;asLastName;[Company]Address;asCompanyAddr)
```

## Example 2

The following example returns the *[Clients]* record numbers in the array *alRecordNumbers* and the *[Clients]Names* field values in the array *asNames*:

```
SELECTION TO ARRAY([Clients];alRecordNumbers;[Clients]Names;asNames)
```

The same example can be written:

```
SELECTION TO ARRAY([Clients];alRecordNumbers;*)
SELECTION TO ARRAY([Clients]Names;asNames;*)
SELECTION TO ARRAY
```

# Size of array

Size of array ( array ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ⇒ | Array whose size is returned |
| Function result | Longint | ⇐ | Returns the number of elements in array |

## Description

The **Size of array** command returns the number of elements in *array*.

## Example 1

The following example returns the size of the array *anArray*:

```
vlSize:=Size of array(anArray) ` vlSize gets the size of anArray
```

## Example 2

The following example returns the number of rows in a two-dimensional array:

```
vlRows:=Size of array(a2DArray) ` vlRows gets the size of a2DArray
```

## Example 3

The following example returns the number of columns for a row in a two-dimensional array:

```
vlColumns:=Size of array(a2DArray{10}) ` vlColumns gets the size of a2DArray{10}
```

# SORT ARRAY

SORT ARRAY ( array {; array2 ; ... ; arrayN}{; > or <} )

| Parameter | Type | | Description |
|---|---|---|---|
| array | Array | ⇒ | Arrays to sort |
| > or < | Operator | ⇒ | ">" to sort in Ascending order, or "<" to sort in Descending order, or Ascending order if omitted |

## Description

The **SORT ARRAY** command sorts one or more arrays into ascending or descending order.

**Note:** You cannot sort *Pointer* or *Picture* arrays. You can sort the elements of a two-dimensional array (i.e., *a2DArray{$vlThisElem}*) but you cannot sort the two-dimensional array itself (i.e., *a2DArray*).

The last parameter specifies whether to sort *array* in ascending or descending order. The “greater than” symbol (>) indicates an ascending sort; the “less than” symbol (<) indicates a descending sort. If you do not specify the sorting order, then the sort is ascending.

If more than one array is specified, the arrays are sorted following the sort order of the first array; no multi-level sorting is performed here. Instead you can use the **MULTI SORT ARRAY** command when you want to sort synchronized arrays.

## Example 1

The following example creates two arrays and then sorts them by company:

```
ALL RECORDS([People])
SELECTION TO ARRAY([People]Name;asNames;[People]Company;asCompanies)
SORT ARRAY(asCompanies;asNames;>)
```

However, because **SORT ARRAY** does not perform multi-level sorts, you will end up with people’s names in random order within each company. To sort people by name within each company, you would write:

```
ALL RECORDS([People])
ORDER BY([People];[People]Company;>;[People]Name;>)
SELECTION TO ARRAY([People]Name;asNames;[People]Company;asCompanies)
```

## Example 2

You display the names from a *[People]* table in a floating window. When you click on buttons present in the window, you can sort this list of names from A to Z or from Z to A. As several people may have the same name, you also can use a *[People]ID number* field, which is indexed unique. When you click in the list of names, you will retrieve the record for the name you clicked. By maintaing a synchronized and hidden array of ID numbers, you are sure to access the record corresponding to the name you clicked:

```
  ` asNames array object method
Case of
   :(Form event=On Load)
      ALL RECORDS([People])
      SELECTION TO ARRAY([People]Name;asNames;[People]ID number;alIDs)
      SORT ARRAY(asNames;alIDs;>)
   :(Form event=On Unload)
      CLEAR VARIABLE(asNames)
      CLEAR VARIABLE(alIDs)
   :(Form event=On Clicked)
      If(asNames#0)
  ` Use the array alIDs to get the right record
         QUERY([People];[People]ID Number=alIDs{asNames})
  ` Do something with the record
      End if
End case
```

```
  ` bA2Z button object method
  ` Sort the arrays in ascending order and keep them synchronized
SORT ARRAY(asNames;alIDs;>)

  ` bZ2A button object method
  ` Sort the arrays in descending order and keep them synchronized
SORT ARRAY(asNames;alIDs;<)
```

# TEXT TO ARRAY

TEXT TO ARRAY ( varText ; arrText ; width ; fontName ; fontSize {; fontStyle {; *}} )

| Parameter | Type | | Description |
|---|---|---|---|
| varText | Text | ⇒ | Original text to be divided |
| arrText | Text array | ⇐ | Array containing the text divided into words or lines |
| width | Longint | ⇒ | Maximum width of string (in pixels) |
| fontName | Text | ⇒ | Name of font |
| fontSize | Longint | ⇒ | Size of font |
| fontStyle | Longint | ⇒ | Style of font |
| * | Operator | ⇒ | If passed = interpret text as multistyle |

## Description

The **TEXT TO ARRAY** command transforms a text variable into a text array. The original varText text (styled or not) is divided and each part becomes an element of the *arrText* array that is returned by the command. This command can be used for example to fill pages or columns with text of a set size.

The original text is divided into "words" based on a line size defined by the command parameters and which takes any styles used into account.

In the *varText* parameter, you pass the text to be divided into array elements. This text may or may not be multistyle. Some parameters are ignored when the text is multistyle.

In the *arrText* parameter, you pass the name of the array to be filled by the divided text.

In the *width* parameter, you pass a size in pixels indicating the maximum line length to measure when dividing the text. For the entire text, the command evaluates the maximum number of words that can "fit" into this width based on the graphic attributes of the text (font, style).

- If it is multistyle text, the styles of the original text are taken into account and the following parameters are ignored if they are passed. In this case, the lines of text in the resulting array keep their original styles (so that they can be printed one by one through a text or string variable for example).
- If it is raw text (no styles), you must pass all the parameters so that the command is able to calculate the length of the lines.

Each array element must contain at least one word. If the *width* passed is too small for the dividing rule to be strictly respected, the array is filled as close as possible according to the parameters and the OK variable is set to 0. For example, if you pass a width of 3 pixels, it is probable that most of the words will be bigger than this length. In this case, the OK variable is set to 0.
This also means that the theoretical maximum size of the array returned is equal to the number of words found in *varText*.

In the *fontName* and *fontSize* parameters, you pass the font name and size with which *varText* must be evaluated by the command in order to divide it. These parameters are mandatory in the case of raw text.

In the *fontStyle* parameter, you pass one or more constants from the **Font Styles** theme:

| Constant | Type | Value |
|---|---|---|
| Bold | Longint | 1 |
| Italic | Longint | 2 |
| Plain | Longint | 0 |
| Underline | Longint | 4 |

This parameter is optional; when it is omitted, the Plain style is used.

The optional * parameter, if passed, forces the *fontName*, *fontSize* and/or *fontStyle* parameters to be taken into account for multistyle text when these parameters are not defined in the original text. However, if these parameters are defined in the original text, then the parameters passed to the command are ignored in all cases.

## Example 1

We want to divide a multistyle text into lines with a maximum size of 200 pixels:

```
TEXT TO ARRAY(theText;TextArray;200;"Arial";20;Plain;*)
 // the Arial, 20, and Normal attributes are only taken into account if they are not defined in the text
```

## Example 2

We want to divide raw text into lines with a maximum size of 350 pixels in Bodoni Bold font, size 14. Since the command does not work correctly if the font is not available, it is important to check for its presence:

```
ARRAY TEXT($FontList;0)
FONT LIST($FontList)
$Font:="Bodoni"
$p:=Find in array($FontList;$Font)
If($p>0)
   TEXT TO ARRAY(theText;TextArray;350;"Bodoni";14;Bold)
Else
 // use another font
End if
```

## Example 3

Multistyle text must be printed without any styles in Arial Normal font, size 12 with a maximum width of 600 pixels:

```
 // we transform the multistyle text into raw text
$RawText:=OBJECT Get plain text(vText)
 // we fill the array
TEXT TO ARRAY($RawText;TextArray;600;"Arial";12)
```

## Example 4

You need to print in a 400-pixel wide area, a text with a maximum of 80 lines and using the largest font possible (without exceeding 24 points). You can write:

```
ARRAY TEXT(TextArray;0)
$Size:=24
Repeat
   TEXT TO ARRAY($RawText;TextArray;400;"Arial";$Size)
   $Size:=$Size-1
   $n:=Size of array(TextArray)
Until($n<=80)
```

# _o_ARRAY STRING

_o_ARRAY STRING ( strLen ; arrayName ; size {; size2} )

| Parameter | Type | | Description |
|---|---|---|---|
| strLen | Longint | ⇒ | Length of string (1... 255) |
| arrayName | Array | ⇒ | Name of the array |
| size | Longint | ⇒ | Number of elements in the array or Number of rows if size2 is specified |
| size2 | Longint | ⇒ | Number of columns in a two-dimensional array |

## Compatibility note

The operation of the **_o_ARRAY STRING** command is strictly identical to that of the **ARRAY TEXT** command (the *strLen* parameter is ignored). It is now recommended to use **ARRAY TEXT** exclusively in your 4D developments.

# Backup

- On Backup Shutdown Database Method
- On Backup Startup Database Method
- BACKUP
- CHECK LOG FILE
- GET BACKUP INFORMATION
- GET RESTORE INFORMATION
- INTEGRATE MIRROR LOG FILE
- Log File
- LOG FILE TO JSON
- New log file
- RESTORE
- SELECT LOG FILE
- *_o_INTEGRATE LOG FILE*

# On Backup Shutdown Database Method

---

The **On Backup Shutdown Database Method** is called every time a database backup ends. The reasons for the stoppage of a backup can be the end of the copy, user interruption or an error.
This concerns all 4D environments: 4D (all modes), 4D Server as well as 4D applications compiled and merged with 4D Volume Desktop.

The **On Backup Shutdown Database Method** allows verifying that the backup was executed correctly. It receives, in the *$1* parameter, a value representing the status of the backup once completed:

- If the backup was executed correctly, *$1* equals 0.
- If the backup was interrupted by the user or following an error, *$1* is different from 0.
  - If the backup was stopped by the **On Backup Startup Database Method** (*$0* # 0), *$1* gets the value actually returned in the *$0* parameter. This allows you to implement a customized error management system.
  - If the backup was stopped due to an error, the error code is returned in *$1*.

  In any case, you can get information about the error using the **GET BACKUP INFORMATION** command.

**Note**: You must declare the *$1* parameter (longint) in the database method:

```
C_LONGINT($1)
```

It is important to note that in the case of an error during backup (disk full, support unavailable, etc.), the information related to the error is only displayed in the 4D Server monitor or in the MSC, and copied into the backup log. No alert dialog box appears and the *error* variable is not modified. If you want to be able to notify the administrator that an error has occurred, particularly in the context of an application running in client/server mode, you will need to use the **On Backup Shutdown Database Method**.

## On Backup Startup Database Method

---

The **On Backup Startup Database Method** is called every time a database backup is about to start (manual backup, scheduled automatic backup, or using the **BACKUP** command).
This concerns all 4D environments: 4D (all modes), 4D Server and databases merged with 4D Volume Desktop.

The **On Backup Startup Database Method** allows verifying that the backup started. In this method, you should return a value that authorizes or refuses the backup in the $0 parameter:

- If $0 = 0, the backup can be launched.
- If $0 # 0, the backup is not authorized. The operation is cancelled and an error is returned. You can get the error using the **GET BACKUP INFORMATION** command.

You can use this database method to verify backup execution conditions (user, date of the last backup, etc.).

**Note:** You must declare the *$0* parameter (longint) in the database method:

```
C_LONGINT($0)
```

# BACKUP

BACKUP
Does not require any parameters

## Description

The **BACKUP** command starts the backup of the database using the current backup settings. No confirmation dialog is displayed; however, a progress bar appears on screen.

Backup settings are set in the Database Settings ("Preferences" in 4D v11 SQL). They are also stored in the Backup.XML file located in the subfolder Preferences/Backup of the database.

The **BACKUP** command calls the **On Backup Startup Database Method** at the beginning of its execution and the **On Backup Shutdown Database Method** at the end of its execution.
Because of this mechanism, the command should not be called from one of these database methods.

**4D Server:** When called from a client machine, **BACKUP** is considered as a stored procedure; it is still executed on the server.

## System variables and sets

If the backup is performed correctly, the system variable OK is set to 1; otherwise, it is set to 0.

## Error management

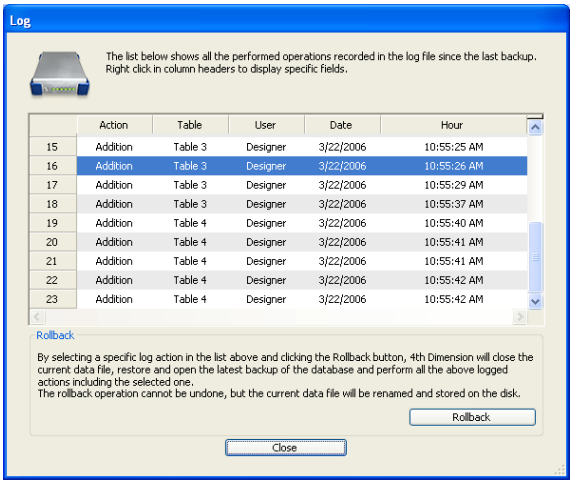
If an incident occurs during backup, information about the incident is written in the backup log and the top-level error is sent only to the **On Backup Shutdown database method**. It is therefore particularly important to use this database method in order to be able to manage back-up related errors using programming.

# CHECK LOG FILE

CHECK LOG FILE
Does not require any parameters

## Description

The **CHECK LOG FILE** command displays the dialog box for viewing the current log file of the database (which can also be accessed via the Maintenance Security Center window):




This dialog box includes the **Rollback** button that can be used to cancel operations carried out on the data of the database. For more information about this dialog box, please refer to the Design Reference manual of 4D.

**Note:** Since the rollback function is relatively powerful, it is recommended that access to the **CHECK LOG FILE** command be restricted to the database administrators.

This command can only be used in the context of single-user applications. More particularly, it allows access to the rollback function from 4D Volume Desktop applications (applications with no Design mode). If it is called within a client/server application, the command has no effect and the error 1421 is returned.

## Error Handling

- If this command is executed in a database operating without a log file, it does nothing and the error 1403 is returned.
- If this command is executed in a client/server database, it does nothing and the error 1421 is returned.
  You can intercept these errors using an error-handling method installed with the **ON ERR CALL** command.

# GET BACKUP INFORMATION

GET BACKUP INFORMATION ( selector ; info1 ; info2 )

| Parameter | Type | | Description |
|---|---|---|---|
| selector | Longint | ➡ | Type of information to get |
| info1 | Longint, Date | ⬅ | Value 1 of the selector |
| info2 | Time, String | ⬅ | Value 2 of the selector |

## Description

The **GET BACKUP INFORMATION** command gets information related to the last backup performed on the database data.

Pass the type of information to get in *selector*. You can use one of the following constants, placed in the "**Backup and Restore**" theme:

| Constant | Type | Value |
|---|---|---|
| Last backup date | Longint | 0 |
| Last backup status | Longint | 2 |
| Next backup date | Longint | 4 |

The type and content of the *info1* and *info2* parameters depend on the value of *selector*.

- If *selector* = 0 (Last Backup Date), *info1* returns the date and *info2* the time of the last backup.
- If *selector* = 2 (Last Backup Status), *info1* returns the number and *info2* the text of the status of the last backup.
- If *selector* = 4 (Next Backup Date), *info1* returns the date and *info2* the time of the next scheduled backup.

# GET RESTORE INFORMATION

GET RESTORE INFORMATION ( selector ; info1 ; info2 )

| Parameter | Type | | Description |
|---|---|---|---|
| selector | Longint | ➡ | Type of information to get |
| info1 | Longint, Date | ⬅ | Value 1 of the selector |
| info2 | String, Time | ⬅ | Value 2 of the selector |

## Description

The **GET RESTORE INFORMATION** command gets information related to the last automatic database restore.

Pass the type of information to get in *selector*. You can use one of the following constants, placed in the "**Backup and Restore**" theme:

| Constant | Type | Value |
|---|---|---|
| Last restore date | Longint | 0 |
| Last restore status | Longint | 2 |

The type and content of the *info1* and *info2* parameters depend on the value of *selector*.

- If *selector* = 0 (Last Restore Date), *info1* returns the date and *info2* the time of the last automatic database restore.
- If *selector* = 2 (Last Restore Status), *info1* returns the number and *info2* the text of the status of the last automatic database restore.

**Note:** This command does not take manual database restores into account.

# INTEGRATE MIRROR LOG FILE

INTEGRATE MIRROR LOG FILE ( pathName ; operationNum {; mode {; errObject}} )

| Parameter | Type | | Description |
|---|---|---|---|
| pathName | Text | ➡ | Name or pathname of the log file to be integrated |
| operationNum | Real variable | ➡ | Number of last operation integrated or -2 to integrate the whole file |
| | | ⬅ | New number of last operation integrated |
| mode | Longint | ➡ | 0=strict mode (default mode), 1=auto repair mode |
| errObject | Object variable | ⬅ | Missed operation(s) |

## Description

**Preliminary note:** This command only works with 4D Server. It can only be executed via the **Execute on server** command or in a stored procedure.

The **INTEGRATE MIRROR LOG FILE** integrates the log file designated by *pathName* into a 4D Server database, after the *operationNum* operation. The command accepts to integrate any log file into the database, even if it does not correspond to the the data file. This command is specifically intended for use in the context of a mirror database.

**Note:** Since 4D v14, it is possible to use a log file as part of a "mirror" database: the "Use Log File" option can now be checked in the Database Settings of a 4D Server used as a logical mirror, thus allowing the implementation of a series of cascading mirror servers (see the **Setting up a logical mirror** section in the 4D Server manual).

Unlike the existing **_o_INTEGRATE LOG FILE** command, at the end of its execution **INTEGRATE MIRROR LOG FILE** does not replace the current log file with the integrated one: the current log file of the database continues to be used. Accordingly, any changes made during integration are saved in the current log file.

In *pathName*, you pass an absolute or relative path to the database folder. If you pass an empty string in this parameter, a standard open file dialog box appears so that you can specify the file to be integrated. If this dialog box is canceled, no file is integrated and the *OK* system variable is set to 0.

In the *operationNum* variable, you pass the number of the last operation integrated, so that integration begins at the next operation. After integration, the value of the *operationNum* variable is updated with the number of the last operation integrated. You must save this variable and then reuse it directly as the *operationNum* parameter for the next integration operation. This allows you to follow on with subsequent log file integrations using **INTEGRATE MIRROR LOG FILE**. Pass -2 in the variable to integrate all the operations in the log file.

***Compatibility note:*** *In versions of 4D prior to v15 R4, the operationNum parameter was optional; however, from now on, if the operationNum parameter is omitted, an error is generated. To restore the original functioning of your former code, you can just pass -2 in the operationNum parameter variable.*

In *mode*, you pass the integration mode you want to activate. You can use one of the following constants found in the "**Backup and Restore**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Auto repair mode | Longint | 1 | Use flexible mode with auto-repair actions and fill the *errObject* parameter (if any) |
| Strict mode | Longint | 0 | Use strict integration mode (default) |

- Strict mode: In this mode, as soon as an error occurs during integration, it stops and you have to use the MSC in order to trace the error. This secure mode is used by default and is recommended in most cases.
- Auto repair mode: In this mode, when a non-critical error occurs, it is bypassed and integration continues. If you passed the *errObject* parameter, each error is logged and can be analyzed afterwards.
  Cases of non-critical errors are:
  - The log requests to add a record, but this record already exists in the data.
    Repair action: 4D updates the record.
  - The log requests to update a record, but this record does not yet exist.
    Repair action: 4D adds the record.
  - The log requests to delete a record, but this record does not exist.
    Repair action: 4D does nothing.

**Note**: In strict mode (default mode), integration will stop at the first error encountered. In this case, if you want to continue with the integration you will need to use the MSC.

When one of these anomalies occurs in auto-repair mode, the record concerned is automatically "repaired" and the related operation is logged in the *errObject* parameter. After execution is completed, the *errObject* parameter lists all the repaired records. It contains a single object array named "operations" built as follows:

```
{"operations":
    [
        {
            "operationType":24,
            "operationName":"Create record",
            "operationNumber":2,
            "contextID":48,
            "timeStamp":"2015-07-10T07:53:02.413Z",
            "dataLen":24,
            "recordNumber":0,
            "tableID":"F4CXXXXX",
            "tableName":"Customers",
            "fields": {
                "1": 9,
                "2": "test value",
                "3": "2003-03-03T00:00:00.000Z",
                "4": "BlobPath: Table 1/Field 4/Data_9ACB28F1A2744FDFA5822B22F18B2E12.png",
                "8": "BlobID: 2"
              }
        },
        {...}
    ]
```

**Warning:** The auto-repair mode must be used in specific cases since it bypasses 4D's internal data integrity checking features. It can be used, for example, when an intermediary log file has been lost or corrupted and you want to recover as many operations as possible. In any case, you need to pay particular attention to data integrity when using this mode.

The actual list of available properties depends on the operation type (i.e.: create record, delete record, modify record, create Blob, etc.). Here are the main properties:

- *operationType*: internal code for the operation
- *operationName*: kind of operation, for example "create record," "modify record"
- *operationNumber*: internal number of operation in the log file
- *contextID*: ID of execution context; the context is detailed in the *extraData* section
- *timeStamp*: timestamp of operation in the log file
- *dataLen*: internal size of data
- *recordNumber*: internal record number
- *tableID*: internal ID of the table
- *tableName*: name of the table
- *fields*: object containing the list of field numbers along with their values. All fields in the table are logged.
  In case of Blob or picture values, different information is provided based on their storage location:
  - If the Blob or picture is stored inside the data file, the property will be "BlobID:"+ an internal Blob number, for example: "BlobID:1"
  - If the Blob or picture is stored outside the data file, the property will be "BlobPath:" + the path for the data, for example: "BlobPath: Table 1/Field 6/Data_EE12D091535F9748BCE62EDE972A4BA2.jpg"
- *extraData*: user context data, including user name and ID, task name and ID, host machine name, and client version.
- *sequenceNumber*: current number within auto-increment sequence.
- *primaryKey*: primary key value.

## Example

You want to integrate a log file on the mirror server in auto-repair mode:

```
 //to be executed on the server
C_OBJECT($err)
C_LONGINT($num) //-2 to integrate all operations
INTEGRATE MIRROR LOG FILE("c:¥¥mirror¥¥logNew.journal";$num;Auto repair mode;$err)
```

## System variables and sets

If the integration is carried out correctly, the system variable OK is set to 1; otherwise, it is set to 0.

# Log File

Log File -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String | ➲ | Long name of the database log file |

## Description

The **Log File** command returns the long name (i.e. the complete pathname of the file, including its name) of the current log file of the open database.

If the database is operating without a log file, the command returns an empty string and the system variable OK is set to 0.

If the database operates with a log file, the system variable OK is set to 1. The pathname returned by the command is expressed with the syntax of the current platform.

**WARNING:** If you execute this command from a 4D Client machine, only the log file name is returned, not the long name

## System variables and sets

- If the database is operating without a log file, the system variable OK is set to 0; otherwise, it is set to 1.
- If for some reason the log file becomes unavailable during a working session, error 1274 is generated and 4D Server does not allow users to write data anymore. When the log file is available again, it is necessary to do a backup.

# LOG FILE TO JSON

LOG FILE TO JSON ( destFolderPath {; maxSize {; logPath {; fieldAtt}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| destFolderPath | Text | ⇒ | Path of the saved file destination folder |
| maxSize | Longint | ⇒ | Maximum size of JSON file to create (bytes) |
| logPath | Text | ⇒ | Pathname of log file to export; use current log file if omitted |
| fieldAtt | Longint | ⇒ | Field description attribute: 1 = use number (default), 2 = use name |

## Description

The **LOG FILE TO JSON** command saves the current log file, or any specified log file, in JSON format.

Once a log (binary file) is saved in JSON, its contents can be read and interpreted by the database administrator or any user in order to analyze database events, for example.

In *destFolderPath*, pass the path of the folder where you want to store the JSON file. This file is named **JournalExport.json**.

By default, the maximum size of the exported JSON file is 10 MB. When this size is reached, the file is closed and a new file is created. Limiting the size of each JSON file reduces memory requirements for analyzing the files. You can change the maximum size for the exported file by setting a value (in bytes) in the *maxSize* parameter. Passing 0 resets to the default size. Passing a negative value removes any size limit.

By default, if the *logPath* parameter is omitted, the command saves the current log file. If you want to export a specific log file, pass its path in the *logPath* parameter. The log file must be a file with the ".journal" extension. If you want to export an archived log file (".4bl" extension), you need to convert it beforehand using the**RESTORE** command. You can pass an empty string ("") to display the standard open file dialog box, allowing the user to select the log file to save. The selected log file path is returned in the **Document** system variable.

**Note:** When the command saves the current log file, the database is not locked. New operations can be executed while the file is being written on disk - these operations will not be included in the saved file.

When you export the current log file, the *fieldAtt* parameter allows you to define how fields will be described in the exported attribute: by number (default), or by name. You can pass one of the following constants, found in the "**Backup and Restore**" constant theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Field attribute with name | Longint | 2 | Fields are identified by their name. Example: {"LastName":"Jones"} |
| Field attribute with number | Longint | 1 | Fields are identified by their number (default if omitted). Example: {"5":"Jones"}. |

**Note:** When you export an external log file, fields are always identified by their number.

The saved JSON file contains all the operations recorded in the log, in the form of an array of JSON objects. Each object contains several properties describing the operation. Example:

```
[
   {
      "operationType":25,
      "operationName":"Modify record",
      "operationNumber":45,
      "contextID":37,
      "timeStamp":"2015-06-11T09:13:17.138Z",
      "dataLen":42,
      "recordNumber":4,
      "tableID":"5AFA15123F991C43B6ACF8B46A914BD0",
      "tableName":"elem",
      "fields": {
            "1": "primkey5",
            "2": -5,
            "5": "data 25"
        },
      "primaryKey": "8"
   },
   {
      "operationType":23,
```

```
      "operationName":"Save seqnum",
      "operationNumber":46,
      "contextID":37,
      "timeStamp":"2015-06-11T09:13:17.138Z",
      "sequenceNumber":23,
      "tableID":"5AFA15123F991C43B6ACF8B46A914BD0",
      "tableName":"elem"
   },
   {
      "operationType":24,
      "operationName":"Create record",
      "operationNumber":47,
      "contextID":37,
      "timeStamp":"2015-06-11T09:13:17.138Z",
      "dataLen":570,
      "recordNumber":7,
      "tableID":"5AFA15123F991C43B6ACF8B46A914BD0",
      "tableName":"elem",
      "fields": {
            "1": 9,
            "2": "test value",
            "3": "2003-03-03T00:00:00.000Z",
            "4": "BlobPath: Table 1/Field 4/Data_9ACB28F1A2744FDFA5822B22F18B2E12.png",
            "8": "BlobID: 2"
      },
      "extraData": {
          "task_id": 1,
          "user_name": "Vanessa Smith",
          "user4d_id": 1,
          "host_name": "iMac-VSmith-0833",
          "task_name": "Application process",
          "client_version": -1610541776
      },
      "primaryKey": "9"
   }
]
```


**Note:** If you passed Field attribute with name in the *fieldAtt* parameter, the "fields" object would look like this:


```
...
      "fields": {
            "ID": 9,
            "Field_2": "test value",
            "Date_Field": "2003-03-03T00:00:00.000Z",
            "Field_4": "BlobPath: Table 1/Field 4/Data_9ACB28F1A2744FDFA5822B22F18B2E12.png",
            "Field_8": "BlobID: 2"
      },...
```


The actual list of available properties depends on the operation type (i.e.: create record, delete record, modify record, create Blob, etc.). Here are the main properties:

- *operationType*: internal code for the operation
- *operationName*: kind of operation, for example "create record," "modify record"
- *operationNumber*: internal number of operation in the log file
- *contextID*: ID of execution context; the context is detailed in the *extraData* section
- *timeStamp*: timestamp of operation in the log file
- *dataLen*: internal size of data
- *recordNumber*: internal record number
- *tableID*: internal ID of the table
- *tableName*: name of the table
- *fields*: object containing the list of field numbers (or field names) along with their values. Any fields with a modified value are logged.
  In case of Blob or picture values, different information is provided based on their storage location:
  - If the Blob or picture is stored inside the data file, the property will be "BlobID:"+ an internal Blob number, for example: "BlobID:1"

    - If the Blob or picture is stored outside the data file, the property will be "BlobPath:" + the path for the data, for example: "BlobPath: Table 1/Field 6/Data_EE12D091535F9748BCE62EDE972A4BA2.jpg"
- *extraData*: user context data, including user name and ID, task name and ID, host machine name, and client version.
- *sequenceNumber*: current number within auto-increment sequence.
- *primaryKey*: primary key value.

## Example

You want to save the current log file in JSON:

```
LOG FILE TO JSON("c:¥¥4Dv15¥¥ExportLogs")
```

You want to save a specific log file in JSON with field names:

```
LOG FILE TO JSON("c:¥¥4Dv15¥¥ExportLogs";0;"c:¥¥4Dv15¥¥Backup¥¥old_myDB.journal";Field attribute with name)
```

## System variables and sets

The **LOG FILE TO JSON** command modifies the value of the OK and Document variables: if the JSON file was correctly saved, OK is set to 1 and Document contains the pathname of the file. If you passed "" in the *logPath* parameter and the user canceled the file selection dialog box, OK is set to 0 and Document contains an empty string. If the user selected an invalid file, OK is set to 0 and Document contains the file path.

# New log file

New log file -> Function result

| Parameter | Type | Description |
| --- | --- | --- |
| Function result | Text | Full pathname of closed log file |

## Description

**Preliminary note:** This command only works with 4D Server. It can only be executed via the **Execute on server** command or in a stored procedure.

The **New log file** command closes the current log file, renames it and creates a new one with the same name in the same location as the previous one. This command is meant to be used for setting up a backup system using a logical mirror (see the section "**Setting up a logical mirror**" in the 4D Server Reference Manual).

The command returns the full pathname (access path + name) of the log file being closed (called the "segment"). This file is stored in the same location as the current log file (specified on the Configuration page in the Backup theme of the Preferences). The command does not carry out any processing (compression, segmentation) on the saved file. No dialog box appears.

The file is renamed with the current backup numbers of the database and of the log file, as shown in the following example: *DatabaseName[BackupNum-LogBackupNum].journal*. For instance:

- If the MyDatabase.4DD database has been saved 4 times, the last backup file will be named *MyDatabase[0004].4BK*. The name of the first "segment" of the log file will therefore be *MyDatabase[0004-0001].journal*.
- If the MyDatabase.4DD database has been saved 3 times and the log file has been saved 5 times since, the name of the 6th backup of the log file will be *MyDatabase[0003-0006].journal*.

## Error management

In the event of an error, the command generates a code that can be intercepted using the **ON ERR CALL** command.

# RESTORE

RESTORE {( archivePath {; destFolderPath} )}

| Parameter | Type | | Description |
|---|---|---|---|
| archivePath | Text | ⇒ | Pathname of archive to restore |
| destFolderPath | Text | ⇒ | Pathname of destination folder |

## Description

The **RESTORE** command can be used to restore the file(s) included in a 4D archive. This command is useful as part of custom interfaces for managing backups.

If you do not pass the *archivePath* parameter, the command displays an open file dialog box so that the user can select the archive to restore.

The *archivePath* parameter lets you indicate the pathname of the archive file to be restored. This pathname must be expressed with the system syntax. You can pass an absolute pathname or a pathname relative to the database structure file. In this case (if the *destFolderPath* parameter is omitted), the standard restore dialog box appears with the archive pre-selected, so that the user can designate the destination folder. When the procedure is completed, a warning dialog box appears and the folder containing the restored elements is displayed.

You can also pass the *destFolderPath* parameter with the pathname of the destination folder of the restored elements. This pathname must be expressed with the system syntax. You can pass an absolute pathname or a pathname relative to the database structure file. If you pass this parameter, a preconfigured restore dialog box appears so that only the user can launch or cancel the restore procedure. When the procedure is completed, the window is simply reclosed without displaying any additional information.

The **RESTORE** command modifies the value of the *OK* and *Document* variables: if the restore was carried out correctly, *OK* is set to 1 and *Document* contains the path of the restoration folder. If the user cancels the restoration dialog box, interrupts the restoration or if an error occurs, *OK* is set to 0 and *Document* contains an empty string. You can intercept the error using a method installed via the **ON ERR CALL** command.

**Note:** In a 4D application that is compiled and merged with 4D Volume Desktop, the **RESTORE** command causes the display of a standard open file dialog box that lists by default any files having the “4BK” extension.

# SELECT LOG FILE

SELECT LOG FILE ( logFile )

| Parameter | Type | | Description |
|---|---|---|---|
| logFile | Operator, String | ⇒ | Name of the log file or "*" for closing the current log file |

## Description

The **SELECT LOG FILE** command creates, or closes the log file according to the value you pass in *logFile*.

**Note:** Calling **SELECT LOG FILE** is the same as selecting/deselecting the **Use Log File** option on the **Backup/Configuration** page of the application Preferences.

In *logFile*, pass the name or the full pathname of the log file to be created. If you only pass a name, the file will be created in the "Logs" folder of the database located next to the database structure file.

If you pass an empty string in *logFile*, **SELECT LOG FILE** presents an Save File dialog box, allowing the user to choose the name and location of the log file to be created. If the file is created correctly, the *OK* variable is set to 1. Otherwise, if the user clicks Cancel or if the log file could not be created, **OK** is set to 0.

**Note:** The new log file is not generated immediately after execution of the command, but rather after the next backup (the parameter is kept in the data file and will be taken into account even if the database is closed in the meantime). You can call the **BACKUP** command to trigger the creation of the log file.

If you pass *"*"* in *logFile*, **SELECT LOG FILE** closes the current log file for the database. The *OK* variable is set to 1 when the log file is closed.

If you use **SELECT LOG FILE** to create a log file when a full backup has not yet been performed and the data file already contains records, 4D then generates an error *-4447,* which you can intercept with an **ON ERR CALL** method.

## System variables and sets

OK is set to 1 if the log file is correctly created, or closed.

## Error management

An error -4447 is generated if the operation cannot be performed because the database needs to be backed up. You can intercept the error with an **ON ERR CALL** method.

# _o_INTEGRATE LOG FILE

_o_INTEGRATE LOG FILE ( pathName )

| Parameter | Type | | Description |
|---|---|---|---|
| pathName | Text | ⇨ | Name or pathname of the log file to be integrated |

## Compatibility note

The **_o_INTEGRATE LOG FILE** command is declared obsolete in 4D beginning with version 16 and is kept only for compatibility reasons. 4D's backup feature using a logical mirror is now based solely on the **INTEGRATE MIRROR LOG FILE** command, which has been optimized and provides greater flexibility.

# BLOB

- BLOB Commands
- BLOB PROPERTIES
- BLOB size
- BLOB TO DOCUMENT
- BLOB to integer
- BLOB to list
- BLOB to longint
- BLOB to real
- BLOB to text
- BLOB TO VARIABLE
- COMPRESS BLOB
- COPY BLOB
- DECRYPT BLOB
- DELETE FROM BLOB
- DOCUMENT TO BLOB
- ENCRYPT BLOB
- EXPAND BLOB
- INSERT IN BLOB
- INTEGER TO BLOB
- LIST TO BLOB
- LONGINT TO BLOB
- REAL TO BLOB
- SET BLOB SIZE
- TEXT TO BLOB
- VARIABLE TO BLOB

# BLOB Commands

## Definition

4D supports the BLOB (Binary Large OBjects) data type.

You can define BLOB fields, BLOB variables and BLOB arrays:

- To create a BLOB field, select BLOB in the **Field type** drop-down-list within the **Field Properties** window.
- To create a BLOB variable, use the compiler declaration command **C_BLOB**. You can create local, process, and interprocess variables of type BLOB.
- To create a BLOB array, use the **ARRAY BLOB** command.

Within 4D, a BLOB is a contiguous series of variable length bytes, which can be treated as one whole object or whose bytes can be addressed individually. A BLOB can be empty (null length) or can contain up to 2147483647 bytes (2 GB).

## BLOBs and Memory

A BLOB is loaded into memory in its entirety. A BLOB variable or BLOB array is held and exists in memory only. A BLOB field is loaded into memory from the disk, like the rest of the record to which it belongs.

Like the other field types that can retain a large amount of data (such as the Picture field type), BLOB fields are not duplicated in memory when you modify a record. Consequently, the result returned by the commands **Old** and **Modified** is not significant when applied to a BLOB field.

## Displaying BLOBs

A BLOB can retain any type of data, so it has no default representation on the screen. If you display a BLOB field or variable in a form, it will always appear blank, whatever its contents.

## BLOB fields

You can use BLOB fields to store any kind of data, up to 2 GB. You cannot index a BLOB field, so you must use a formula in order to search records on values stored in a BLOB field.

## Parameter passing, Pointers and function results

4D BLOBs can be passed as parameters to 4D commands or plug-in routines that expect a BLOB parameters. BLOBS can also be passed as parameters to a user method or be returned as a function result.

To pass a BLOB to your own methods, you can also define a pointer to the BLOB and pass the pointer as parameter.

Examples:

```
  ` Declare a variable of type BLOB
C_BLOB(anyBlobVar)
  ` The BLOB is passed as parameter to a 4D command
SET BLOB SIZE(anyBlobVar;1024*1024)
  ` The BLOB is passed as parameter to an external routine
$errCode:=Do Something With This BLOB(anyBlobVar)
  ` The BLOB is passed as a parameter to a method that returns a BLOB
C_BLOB(retrieveBlob)
retrieveBlob:=Fill_Blob(anyBlobVar)
  ` A pointer to the BLOB is passed as parameter to a user method
COMPUTE BLOB(->anyBlobVar)
```

**Note for Plug-in developers**: A BLOB parameter is declared as “&O” (the letter “O”, not the digit “0”).

## Assignment

You can assign BLOBs to each other.

Example:

```
  ` Declare two variables of type BLOB
C_BLOB(vBlobA;vBlobB)
  ` Set the size of the first BLOB to 10K
SET BLOB SIZE(vBlobA;10*1024)
  ` Assign the first BLOB to the second one
vBlobB:=vBlobA
```

However, no operator can be applied to BLOBs; there is no expression of type BLOB.

## Addressing BLOB contents

You can address each byte of a BLOB individually using the curly brackets symbols *{...}*. Within a BLOB, bytes are numbered from *0* to *N-1*, where *N* is the size of the BLOB. Example:

```
  ` Declare a variable of type BLOB
C_BLOB(vBlob)
  ` Set the size of the BLOB to 256 bytes
SET BLOB SIZE(vBlob;256)
  ` The loop below initializes the 256 bytes of the BLOB to zero
For(vByte;0;BLOB size(vBlob)-1)
   vBlob{vByte}:=0
End for
```

Because you can address all the bytes of a BLOB individually, you can actually store whatever you want in a BLOB field or variable.

## BLOBs 4D commands

4D provides the following commands for working BLOBS:

- **SET BLOB SIZE** resizes a BLOB field or variable.
- **BLOB size** returns the size of a BLOB.
- **DOCUMENT TO BLOB** and **BLOB TO DOCUMENT** enable you to load and write a whole document to and from a BLOB (optionally, the data and resource forks on Macintosh).
- **VARIABLE TO BLOB** and **BLOB TO VARIABLE** as well as **LIST TO BLOB** and **BLOB to list** allow you to store and retrieve 4D variables in BLOBs.
- **COMPRESS BLOB**, **EXPAND BLOB** and **BLOB PROPERTIES** allow you to work with compressed BLOBs
- The commands **BLOB to integer**, **BLOB to longint**, **BLOB to real**, **BLOB to text**, **INTEGER TO BLOB**, **LONGINT TO BLOB**, **REAL TO BLOB** and **TEXT TO BLOB** enable you to manipulate any structured data coming from disk, resources, OS, and so on.
- **DELETE FROM BLOB**, **INSERT IN BLOB** and **COPY BLOB** allow quick handling of large chunks of data within BLOBs.
- **ENCRYPT BLOB** and **DECRYPT BLOB** allow you to encrypt and decrypt data in a 4D database.

These commands are described in this chapter.

In addition:

- **C_BLOB** declares a variable of type BLOB. Refer to the **Compiler** chapter for more information.
- **ARRAY BLOB** creates or resizes a BLOB type array (refer to the **Arrays** section).
- **GET PASTEBOARD DATA** and **APPEND DATA TO PASTEBOARD** enable you to deal with any data type stored in the pasteboard. Refer to the **Managing Pasteboards** section for more information.
- **GET RESOURCE** and **_o_SET RESOURCE** enable you to work with any type stored of resource stored on disk. Refer to the **Resources** chapter for more information.
- **WEB SEND BLOB** enable you to send any type of data to a Web browser. Refer to the **Web Server** chapter for more information.
- **PICTURE TO BLOB**, **BLOB TO PICTURE** and **PICTURE TO GIF** allow you to open and convert pictures. Refer to the **Pictures** chapter for more information.
- **GENERATE ENCRYPTION KEYPAIR** and **GENERATE CERTIFICATE REQUEST** are encryption commands used by the SSL (Secured Socket Layer) secured connection protocol. Refer to the **Using TLS Protocol** section for more information.

# BLOB PROPERTIES

BLOB PROPERTIES ( blob ; compressed {; expandedSize {; currentSize}} )

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ⇒ | BLOB for which to get information |
| compressed | Longint | ⇐ | 0 = BLOB is not compressed, 1 = Compact compression, 2 = Fast compression, -1 = GZIP Best compression, -2 = GZIP Fast compression |
| expandedSize | Longint | ⇐ | Size of BLOB (in bytes) when not compressed |
| currentSize | Longint | ⇐ | Current size of BLOB (in bytes) |

## Description

The **BLOB PROPERTIES** command returns information about the BLOB *blob*.

The *compressed* parameter returns a value indicating if and how the BLOB is compressed. You can compare this value with the following constants, found in the **BLOB** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Compact compression mode | Longint | 1 | Compressed as much as possible (at the expense of the speed of compression and decompression operations). Default method. |
| Fast compression mode | Longint | 2 | Compressed as fast as possible (and will be decompressed as fast as possible), at the expense of the compression ratio (the compressed BLOB will be bigger). |
| GZIP best compression mode | Longint | -1 | Most compact GZIP compression |
| GZIP fast compression mode | Longint | -2 | Fastest GZIP compression |
| Is not compressed | Longint | 0 | No compression |

Whatever the compression status of the BLOB, the *expandedSize* parameter returns the size of the BLOB when it is not compressed.

The parameter *currentSize* returns the current size of the BLOB. If the BLOB is compressed, you will usually obtain *currentSize* less than *expandedSize*. If the BLOB is not compressed, you will always obtain *currentSize* equal to *expandedSize*.

## Example 1

See examples for the commands **COMPRESS BLOB** and **EXPAND BLOB**.

## Example 2

After a BLOB has been compressed, the following project method obtains the percentage of space saved by the compression:

```
   ` Space saved by compression project method
   ` Space saved by compression (Pointer {; Pointer } ) -> Long integer
   ` Space saved by compression ( -> BLOB {; -> savedBytes } ) -> Percentage

C_POINTER($1;$2)
C_LONGINT($0;$vlCompressed;$vlExpandedSize;$vlCurrentSize)

BLOB PROPERTIES($1->;$vlCompressed;$vlExpandedSize;$vlCurrentSize)
If($vlExpandedSize=0)
   $0:=0
   If(Count parameters>=2)
```

```
      $2->:=0
   End if
Else
   $0:=100-(($vlCurrentSize/$vlExpandedSize)*100)
   If(Count parameters>=2)
      $2->:=$vlExpandedSize-$vlCurrentSize
   End if
End if
```

After this method has been added to your application, you can use it this way:

```
  ```
COMPRESS BLOB(vxBlob)
$vlPercent:=Space saved by compression(->vxBlob;->vlBlobSize)
ALERT("The compression saved "+String(vlBlobSize)+" bytes, so "+String($vlPercent;"#0%")+
" of space.")
```
```

# BLOB size

BLOB size ( blob ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ⇒ | BLOB field or variable |
| Function result | Longint | ⇐ | Size in bytes of the BLOB |

## Description

**BLOB size**returns the size of *blob* expressed in bytes.

## Example

The line of code adds 100 bytes to the BLOB *myBlob*:

```
SET BLOB SIZE(BLOB size(myBlob)+100)
```

# BLOB TO DOCUMENT

BLOB TO DOCUMENT ( document ; blob {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Name of the document |
| blob | BLOB | ⇒ | New contents for the document |
| * | Operator | ⇒ | On Macintosh only: Resource fork is written if * is passed; otherwise, Data fork is written |

## Description

**BLOB TO DOCUMENT** rewrites the whole contents of *document* using the data stored in *blob*. You can pass the name of a document in *document*. If the *document* does not exist, the command creates it. If you pass the name of an existing document, make sure that it is not already open, otherwise an error is generated. If you want to let the user choose the document, use the commands **Open document** or **Create document** and use the process variable *document* (see example).

**Notes regarding Macintosh:**

- Macintosh documents can be composed of two forks: the Data fork and the Resource fork. By default, the command **BLOB TO DOCUMENT** rewrites the Data fork of the document. To rewrite the Resource fork of the document instead, pass the optional * parameter. On Windows, the optional * parameter is ignored.
- The documents generated by this command do not have a "type". If you want to create a document with a type, you must use the **SET DOCUMENT TYPE** command.

## Example

You write an Information System that enables you to quickly store and retrieve documents. In a data entry form, you create a button which allows you to save a document that will contain the data previously loaded into a BLOB field. The method for this button could be:

```
$vhDocRef:=Create document("") ` Save the document of your choice
If(OK=1) ` If a document has been created
   CLOSE DOCUMENT($vhDocRef) ` We don't need to keep it open
   BLOB TO DOCUMENT(Document;[YourTable]YourBLOBField) ` Write the document contents
   If(OK=0)
   ` Handle error
   End if
End if
```

## System variables and sets

OK is set to 1 if the document is correctly written, otherwise OK is set to 0 and an error is generated.

## Error Handling

- If you try to rewrite a document that does not exist or that is already open by another process or application, the appropriate File Manager error is generated.
- The disk space may be insufficient for writing the new contents of the document.
- I/O errors can occur while writing the document.

In all cases, you can trap the error using an **ON ERR CALL** interruption method.

# BLOB to integer

BLOB to integer ( blob ; byteOrder {; offset} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ➡ | BLOB from which to get the integer value |
| byteOrder | Longint | ➡ | 0 Native byte ordering 1 Macintosh byte ordering 2 PC byte ordering |
| offset | Variable | ➡ | Offset within the BLOB (expressed in bytes) |
| | | ⬅ | New offset after reading |
| Function result | Integer | ↩ | 2-byte Integer value |

## Description

The **BLOB to integer** command returns a 2-byte Integer value read from the BLOB *blob*.

The *byteOrder* parameter fixes the byte ordering of the 2-byte Integer value to be read. You pass one of the following predefined constants provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Macintosh byte ordering | Longint | 1 |
| Native byte ordering | Longint | 0 |
| PC byte ordering | Longint | 2 |

**Note regarding Platform Independence:** If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage byte swapping issues when using this command.

If you specify the optional *offset* variable parameter, the 2-byte Integer value is read at the offset (starting from zero) within the BLOB. If you do not specify the optional *offset* variable parameter, the first two bytes of the BLOB are read.

**Note:** You should pass an offset (in bytes) value between *0* (zero) and the size of the BLOB minus 2. If you do not do so, an error -111 is generated.

After the call, the variable is incremented by the number of bytes read, Therefore, you can reuse that same variable with another BLOB reading command to read another value.

## Example

The following example reads 20 Integer values from a BLOB, starting at the offset 0x200:

```
$vlOffset:=0x200
For($viLoop;0;19)
   $viValue:=BLOB to integer(vxSomeBlob;PC byte ordering;$vlOffset)
   ` Do something with $viValue
End for
```

# BLOB to list

BLOB to list ( blob {; offset} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ➡ | BLOB containing a hierarchical list |
| offset | Longint | ➡ | Offset within the BLOB (expressed in bytes) |
| | | ⬅ | New offset after reading |
| Function result | ListRef | ↩ | Reference to newly created list |

## Description

The **BLOB to list** command creates a new hierarchical list with the data stored within the BLOB *blob* at the byte offset (starting at zero) specified by *offset* and returns a List Reference number for that new list.

The BLOB data must be consistent with the command. Typically, you will use BLOBs that you previously filled out using the command **LIST TO BLOB**.

If you do not specify the optional *offset* parameter, the list data is read starting from the beginning of the BLOB. If you deal with a BLOB in which several variables or lists have been stored, you must pass the *offset* parameter and, in addition, you must pass a numeric variable. Before the call, set this numeric variable to the appropriate offset. After the call, that same numeric variable returns the offset of the next variable stored within the BLOB.

After the call, if the hierarchical list has been successfully created, the OK variable is set to 1. If the operation could not be performed, the OK variable is set to 0; for example, if there was not enough memory.

**Note regarding Platform Independence: BLOB to list** and **LIST TO BLOB** use a 4D internal format for handling lists stored in BLOBs. As a benefit, you do not need to worry about byte swapping between platforms when using these two commands. In other words, a BLOB created on Windows using those two commands can be reused on Macintosh and vice-versa.

## Example

In this example, the form method for a data entry form extracts a list from a BLOB field before the form appears on the screen, and stores it back to the BLOB field if the data entry is validated:

```
   ` [Things To Do];"Input" Form Method

Case of

   :(Form event=On Load)
      hList:=BLOB to list([Things To Do]Other Crazy Ideas)
      If(OK=0)
         hList:=New list
      End if

   :(Form event=On Unload)
      CLEAR LIST(hList;*)

   :(bValidate=1)
      LIST TO BLOB(hList;[Things To Do]Other Crazy Ideas)

End case
```

## System variables and sets

The OK variable is set to 1 if the list has been successfully created, otherwise it is set to 0.

# BLOB to longint

BLOB to longint ( blob ; byteOrder {; offset} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ➡ | BLOB from which to get the Long Integer value |
| byteOrder | Longint | ➡ | 0 = Native byte ordering, 1 = Macintosh byte ordering, 2 = PC byte ordering |
| offset | Variable | ➡ | Offset within the BLOB (expressed in bytes) |
| | | ⬅ | New offset after reading |
| Function result | Longint | ↩ | 4-byte Long Integer value |

## Description

The **BLOB to longint** command returns a 4-byte Long Integer value read from the BLOB *blob*.

The *byteOrder* parameter fixes the byte ordering of the 4-byte Long Integer value to be read. You pass one of the following predefined constants provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Macintosh byte ordering | Longint | 1 |
| Native byte ordering | Longint | 0 |
| PC byte ordering | Longint | 2 |

**Note regarding Platform Independence:** If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage byte swapping issues while using this command.

If you specify the optional *offset* variable parameter, the 4-byte Long Integer is read at the offset (starting from zero) within the BLOB. If you do not specify the optional *offset* variable parameter, the first four bytes of the BLOB are read.

**Note:** You should pass an offset value between *0* (zero) and the size of the BLOB minus 4. If you do not do so, an error -111 is generated.

After the call, the variable is incremented by the number of bytes read. Therefore, you can reuse that same variable with another BLOB reading command to read another value.

## Example

The following example reads 20 Long Integer values from a BLOB, starting at the offset 0x200:

```
$vlOffset:=0x200
For($viLoop;0;19)
   $vlValue:=BLOB to longint(vxSomeBlob;PC byte ordering;$vlOffset)
   ` Do something with $vlValue
End for
```

# BLOB to real

BLOB to real ( blob ; realFormat {; offset} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ➡ | BLOB from which to get the Real value |
| realFormat | Longint | ➡ | 0=Native real format, 1=Extended real format, 2=Macintosh Double real format, 3=Windows Double real format |
| offset | Variable | ➡ | Offset within the BLOB (expressed in bytes) |
| | | ⬅ | New offset after reading |
| Function result | Real | ↩ | Real value |

## Description

The **BLOB to real** command returns a Real value read from the BLOB *blob*.

The *realFormat* parameter fixes the internal format and byte ordering of the Real value to be read. You pass one of the following predefined constants provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Extended real format | Longint | 1 |
| Macintosh double real format | Longint | 2 |
| Native real format | Longint | 0 |
| PC double real format | Longint | 3 |

**Note regarding Platform Independence:** If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage real formats and byte swapping issues while using this command.

If you specify the optional *offset* variable parameter, the Read value is read at the offset (starting from zero) within the BLOB. If you do not specify the optional *offset* variable parameter, the first 8 or 10 bytes of the BLOB are read.

**Note:** You should pass an offset value between *0* (zero) and the size of the BLOB minus 8 or 10. If you do not do so, an error -111 is generated.

After the call, the variable is incremented by the number of bytes read. Therefore, you can reuse that same variable with another BLOB reading command to read another value.

## Example

The following example reads 20 Real values from a BLOB, starting at the offset 0x200:

```
$vlOffset:=0x200
For($viLoop;0;19)
   $vrValue:=BLOB to real(vxSomeBlob;PC byte ordering;$vlOffset)
   ` Do something with $vrValue
End for
```

# BLOB to text

BLOB to text ( blob ; textFormat {; offset {; textLength}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ⇒ | BLOB from which to get the text |
| textFormat | Longint | ⇒ | Format and character set of text |
| offset | Variable | ⇒ | Offset within the BLOB (expressed in bytes) |
| | | ⇐ | New offset after reading |
| textLength | Longint | ⇒ | Number of characters to be read |
| Function result | Text | ⇐ | Text extracted |

## Description

The **BLOB to text** command returns a Text value read from the BLOB *blob*.

The *textFormat* parameter fixes the internal format and character set of the text value to be read. In databases created beginning with version 11, 4D uses the Unicode character set (UTF8) by default for managing text. For the sake of compatibility, this command can be used to "force" conversion using the Mac Roman character set (used in previous versions of 4D). The character set is chosen via the *textFormat* parameter. To do this, pass one of the following constants (found in the **BLOB** theme) in the *textFormat* parameter:

| Constant | Type | Value |
|---|---|---|
| Mac C string | Longint | 0 |
| Mac Pascal string | Longint | 1 |
| Mac text with length | Longint | 2 |
| Mac text without length | Longint | 3 |
| UTF8 C string | Longint | 4 |
| UTF8 text with length | Longint | 5 |
| UTF8 text without length | Longint | 6 |

**Notes:**

- The "UTF8" constants can only be used when the application runs in Unicode mode.
- The "Mac" constants cannot work with texts greater than 32 KB.
- If you want to work with character sets other than UTF8, use the **Convert to text** command.

For more information about these constants and the formats they represent, please refer to the description of the **TEXT TO BLOB** command.

**WARNING:** The number of characters to be read is determined by the *textFormat* parameter, EXCEPT for the formats Mac Text without length and UTF8 Text without length, for which you need to specify the number of characters to be read in the *textLength* parameter. For the other formats, *textLength* is ignored and you can omit it.

If you specify the optional *offset* variable parameter, the Text value is read at the offset (starting from zero) within the BLOB. If you do not specify the optional *offset* variable parameter, the beginning of the BLOB is read according to the value you pass in *textFormat*. Note that you must pass the *offset* variable parameter when you are reading text without length.

**Note:** You should pass an offset value between *0* (zero) and the size of the BLOB minus the size of the text to be read. If you do not do so, the function result is unpredictable.

After the call, the variable is incremented by the number of bytes read. Therefore, you can reuse that same variable with another BLOB reading command to read another value.

# BLOB TO VARIABLE

BLOB TO VARIABLE ( blob ; variable {; offset} )

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ⇒ | BLOB containing 4D variables |
| variable | Variable | ⇐ | Variable to write with BLOB contents |
| offset | Longint | ⇒ | Position of variable within BLOB |
| | | ⇐ | Position of following variable within BLOB |

## Description

The **BLOB TO VARIABLE** command rewrites the variable *variable* with the data stored within the BLOB *blob* at the byte offset (starting at zero) specified by *offset*.

The BLOB data must be consistent with the destination variable. Typically, you will use BLOBs that you previously filled out using the command **VARIABLE TO BLOB**.

If you do not specify the optional *offset* parameter, the variable data is read starting from the beginning of the BLOB. If you deal with a BLOB in which several variables have been stored, you must pass the *offset* parameter and, in addition, you must pass a numeric variable. Before the call, set this numeric variable to the appropriate offset. After the call, that same numeric variable returns the offset of the next variable stored within the BLOB.

**Note: BLOB TO VARIABLE** supports object variables of the **C_OBJECT** type. For more information, refer to the **VARIABLE TO BLOB** command.

After the call, if the variable has been successfully rewritten, the OK variable is set to 1. If the operation could not be performed, the OK variable is set to 0; for example, if there was not enough memory.

**Note regarding Platform Independence: BLOB TO VARIABLE** and **VARIABLE TO BLOB** use a 4D internal format for handling variables stored in BLOBs. As a benefit, you do not need to worry about byte swapping between platforms while using these two commands. In other words, a BLOB created on Windows using either of these commands can be reused on Macintosh, and vice-versa.

## Example

See the examples for the **VARIABLE TO BLOB** command.

## System variables and sets

The OK variable is set to 1 if the variable has been successfully rewritten, otherwise it is set to 0.

# COMPRESS BLOB

COMPRESS BLOB ( blob {; compression} )

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ➡ | BLOB to compress |
| compression | Longint | ➡ | If not omitted: 1, compress as compact as possible 2, compress as fast as possible |

## Description

The **COMPRESS BLOB** command compresses the BLOB *blob* using a compression algorithm. This command only compresses BLOB whose size is over 255 bytes.

The optional *compression* parameter allows to set the way the BLOB will be compressed. You can pass one of the following constants, placed in the **BLOB** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Compact compression mode | Longint | 1 | Compressed as much as possible (at the expense of the speed of compression and decompression operations). Default method. |
| Fast compression mode | Longint | 2 | Compressed as fast as possible (and will be decompressed as fast as possible), at the expense of the compression ratio (the compressed BLOB will be bigger). |
| GZIP best compression mode | Longint | -1 | Most compact GZIP compression |
| GZIP fast compression mode | Longint | -2 | Fastest GZIP compression |

If you pass another value or if you omit the *compression* parameter, compression mode 1 is used (compact internal compression).

**Note:** This command only compresses BLOBs that are greater than or equal to 255 bytes.

After the call, the OK variable is set to 1 if the BLOB has been successfully compressed. If the compression could not be performed, the OK variable is set to 0. If the compression could not be performed because of a lack of memory or because the actual size of the blob is less than 255 bytes, no error is generated and the method resumes its execution.
In any other cases (i.e. the BLOB is damaged), the error -10600 is generated. This error can be trapped using the **ON ERR CALL** command.

After a BLOB has been compressed, you can expand it using the **EXPAND BLOB** command.

To detect if a BLOB has been compressed, use the **BLOB PROPERTIES** command.

**WARNING:** A compressed BLOB is still a BLOB, so there is nothing to stop you from modifying its contents. However, if you do so, the **EXPAND BLOB** command will not be able to decompress the BLOB properly.

## Example 1

This example tests if the BLOB *vxMyBlob* is compressed, and, if it is not, compresses it:

```
BLOB PROPERTIES(vxMyBlob;$vlCompressed;$vlExpandedSize;$vlCurrentSize)
If($vlCompressed=Is not compressed)
   COMPRESS BLOB(vxMyBlob)
End if
```

Note however, that if you apply **COMPRESS BLOB** to an already compressed BLOB, the command detects it and does nothing.

## Example 2

This example allows you to select a document and then compress it:

```
$vhDocRef :=Open document("")
If(OK=1)
   CLOSE DOCUMENT($vhDocRef)
   DOCUMENT TO BLOB(Document;vxBlob)
   If(OK=1)
      COMPRESS BLOB(vxBlob)
      If(OK=1)
         BLOB TO DOCUMENT(Document;vxBlob)
      End if
   End if
End if
```

## Example 3

Sending of raw HTTP data compressed with GZIP:

```
COMPRESS BLOB($blob;GZIP Best compression mode)
C_TEXT($vEncoding)
$vEncoding:="Content-encoding: gzip"
WEB SET HTTP HEADER($vEncoding)
WEB SEND RAW DATA($blob ;*)
```

## System variables and sets

The OK variable is set to 1 if the BLOB has been successfully compressed; otherwise, it is set to 0.

# COPY BLOB

COPY BLOB ( srcBLOB ; dstBLOB ; srcOffset ; dstOffset ; len )

| Parameter | Type | | Description |
|---|---|---|---|
| srcBLOB | BLOB | ⇒ | Source BLOB |
| dstBLOB | BLOB | ⇒ | Destination BLOB |
| srcOffset | Longint | ⇒ | Source position for the copy |
| dstOffset | Longint | ⇒ | Destination position for the copy |
| len | Longint | ⇒ | Number of bytes to be copied |

## Description

The **COPY BLOB** command copies the number of bytes specified by *len* from the BLOB *srcBLOB* to the BLOB *dstBLOB*.

The copy starts at the position (expressed relative to the beginning of the source BLOB) specified by *srcOffset* and takes place at the position (expressed relative to the beginning of the destination BLOB) specified by *dstOffset*.

**Note:** The destination BLOB can be resized if necessary.

# DECRYPT BLOB

DECRYPT BLOB ( toDecrypt ; sendPubKey {; recipPrivKey} )

| Parameter | Type | | Description |
|---|---|---|---|
| toDecrypt | BLOB | ⇒ | Data to decrypt |
| | | ⇐ | Decrypted data |
| sendPubKey | BLOB | ⇒ | Sender's public key |
| recipPrivKey | BLOB | ⇒ | Recipient's private key |

## Description

The **DECRYPT BLOB** command decrypts the content of the BLOB *toDecrypt* using the sender's public key *sendPubKey* and, optionally, the recipient's private key *recipPrivKey*.

The BLOB containing the sender's public key is passed in the *sendPubKey* parameter. This key has been generated by the sender using the **GENERATE ENCRYPTION KEYPAIR** command and it has to be sent to the recipient.

The BLOB containing the recipient's private key can be passed in the optional parameter *recipPrivKey*. In this case, the recipient has to generate a pair of encryption keys with the **GENERATE ENCRYPTION KEYPAIR** command and has to send his/her public key to the sender. The keypair-based encryption system guarantees that the message has been encrypted by the sender only and it can be decrypted by the recipient only. For more information about the keypair-based encryption system, refer to the routine **ENCRYPT BLOB**.

The command **DECRYPT BLOB** offers a checksum functionality in order to avoid any BLOB content modification (deliberate or not). If the encrypted BLOB is damaged or modified, the command will do nothing and an error will be returned.

## Example

Refer to the examples given for the **ENCRYPT BLOB** command.

# DELETE FROM BLOB

DELETE FROM BLOB ( blob ; offset ; len )

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ⇒ | BLOB from which to delete bytes |
| offset | Longint | ⇒ | Starting offset where bytes will be deleted |
| len | Longint | ⇒ | Number of bytes to be deleted |

## Description

The **DELETE FROM BLOB** command deletes the number of bytes specified by *len* from the BLOB *blob* at the position specified by *offset* (expressed relative to the beginning of the BLOB). The BLOB then becomes *len* bytes smaller.

# DOCUMENT TO BLOB

DOCUMENT TO BLOB ( document ; blob {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Name of the document |
| blob | BLOB | ⇒ | BLOB field or variable to receive the document |
| | | ⇐ | Document contents |
| * | Operator | ⇒ | On Macintosh only: Resource fork is loaded if * is passed otherwise Data fork is loaded |

## Description

**DOCUMENT TO BLOB** loads the whole contents of *document* into *blob*. You must pass the name of an existing document that is not already open, otherwise an error will be generated. To let the user choose the document to be loaded into the BLOB, use the command **Open document** and the process variable *document* (see Example).

**Note regarding Macintosh:** Macintosh documents can be composed of two forks: the Data fork and the Resource fork. By default, the command **DOCUMENT TO BLOB** loads the Data fork of the document. To load the Resource fork of the document instead, pass the optional * parameter. On Windows, the optional * parameter is ignored. Note that the 4D environment provides the equivalent of Mac OS resource forks on Windows. For example, the data fork of a 4D database is stored in a file with the file extension .4DB; the resource fork is stored in a file with the same name and the file extension .RSR. On Windows, if you write a 4D application with the data fork and resource fork stored in BLOBs, you just need to access the file corresponding to the fork with which you want to work.

## Example

You write an Information System that enables you to quickly store and retrieve documents. In a data entry form, you create a button that allows you to load a document into a BLOB field. The method for this button could be:

```
$vhDocRef:=Open document("") ` Select the document of your choice
If(OK=1) ` If a document has been chosen
   CLOSE DOCUMENT($vhDocRef) ` We don't need to keep it open
   DOCUMENT TO BLOB(Document;[YourTable]YourBLOBField) ` Load the document
   If(OK=0)
   ` Handle error
   End if
End if
```

## System variables and sets

OK is set to 1 if the document is correctly loaded, otherwise OK is set to 0 and an error is generated.

## Error Handling

- If you try to load (into a BLOB) a document that does not exist or that is already open by another process or application, the appropriate File Manager error is generated.
- An I/O error can occur if the document is locked, located on a locked volume, or if there is problem in reading the document.
- If there is not enough memory to load the document, an error -108 is generated.

In each case, you can trap the error using an **ON ERR CALL** interruption method.

# ENCRYPT BLOB

ENCRYPT BLOB ( toEncrypt ; sendPrivKey {; recipPubKey} )

| Parameter | Type | | Description |
|---|---|---|---|
| toEncrypt | BLOB | ⇒ | Data to encrypt |
| | | ⇐ | Encrypted data |
| sendPrivKey | BLOB | ⇒ | Sender's private key |
| recipPubKey | BLOB | ⇒ | Recipient's public key |

## Description

The **ENCRYPT BLOB** command encrypts the content of the *toEncrypt* BLOB with the sender's private key *sendPrivKey*, as well as optionally the recipient's public key *recipPubKey*. These keys should be generated by the command **GENERATE ENCRYPTION KEYPAIR** (within the "Secured Protocol" theme).

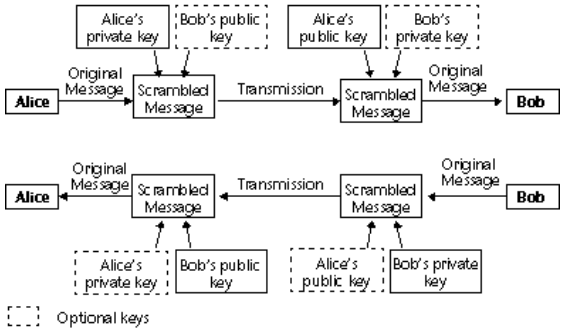
**Note:** This command uses the SSL protocol algorithm and encryption features. To be able to use this command, make sure that the components necessary to the SSL protocol are installed properly on your machine — even though you do not want to use SSL for 4D Web server connections. For detailed information on this protocol, please refer to section **Using TLS Protocol**.

- If one key is used for the encryption (the sender's private key), only people in possession of the public key will be able to read the information. This system guarantees that the sender himself has encrypted the information.
- The simultaneous use of the sender's private key and recipient's public key guarantees that only one recipient will be able to read the information.

The BLOB containing the keys has a PKCS internal format. This standard cross platform format allows exchanging or handling keys simply by copy-pasting in an Email or a text file.

Once the command has been run, the *toEncrypt* BLOB contains the encrypted data that will be decrypted only with the **DECRYPT BLOB** command, with the sender's public key passed as parameter.
Moreover, if the optional recipient's public key has been used to encrypt the information, the recipient's private key will also be necessary for decrypting.

**Encryption principle with public and private keys for message exchange between two people, "Alice" and "Bob":**




**Note:** The cipher contains a checksum functionality in order to avoid any BLOB content modification (deliberately or not). Consequently, an encrypted BLOB should not be modified otherwise it might not be decrypted.

### Optimizing Encryption Commands

Data encryption slows down the execution of your applications, especially if a pair of keys is used. However, you can consider the following optimization tips:

- Depending on the current available memory, the command will execute in "synchronous" or "asynchronous" mode. The asynchronous mode is faster, since it does not freeze the other processes. This mode is automatically used if the available memory is at least twice the size of the data to encrypt.
  Otherwise, for security reasons, the synchronous mode is used. This mode is slower since it freezes the other processes.
- Regarding large BLOBs, you can encrypt only a small "strategic" part of the BLOB in order to reduce the size of data to be processed as well as the processing time.

## Example

- **Using a single key**

A company wants to keep the data stored in a 4D database private. It has to regularly send these information to its subsidiaries through files, via the Internet.

1) The company generates a pair of keys with the command **GENERATE ENCRYPTION KEYPAIR**:

```
 `Method GENERATE_KEYS_TXT
C_BLOB($BPublicKey;$BPrivateKey)
GENERATE ENCRYPTION KEYPAIR($BPrivateKey;$BPublicKey)
BLOB TO DOCUMENT("PublicKey.txt";$BPublicKey)
BLOB TO DOCUMENT("PrivateKey.txt";$BPrivateKey)
```

2) The company keeps the private key and sends a copy of the document containing the public key to each subsidiary. For maximum security, the key should be copied on a disk handed over to the subsidiaries.

3) Then the company copies the private information (stored in the text field, for example) in BLOBs which will be encrypted with the private key:

```
 `Method ENCRYPT_INFO
C_BLOB($vbEncrypted;$vbPrivateKey)
C_TEXT($vtEncrypted)

$vtEncrypted:=[Private]Info
VARIABLE TO BLOB($vtEncrypted;$vbEncrypted)
DOCUMENT TO BLOB("PrivateKey.txt";$vbPrivateKey)
If(OK=1)
   ENCRYPT BLOB($vbEncrypted;$vbPrivateKey)
   BLOB TO DOCUMENT("Update.txt";$vbEncrypted)
End if
```

4) The update files can be sent to the subsidiaries (though a non-secured channel such as the Internet). If a third person gets hold of the encrypted file, he will not be able to decrypt it without the public key.

5) Each subsidiary can decrypt the document with the public key:

```
 `Method DECRYPT_INFO
C_BLOB($vbEncrypted;$vbPublicKey)
C_TEXT($vtDecrytped)
C_TIME($vtDocRef)

ALERT("Please select an encrypted document.")
$vtDocRef:=Open document("") `Select Update.txt
If(OK=1)
   CLOSE DOCUMENT($vtDocRef)
   DOCUMENT TO BLOB(Document;$vbEncrypted)
   DOCUMENT TO BLOB("PublicKey.txt";$vbPublicKey)
   If(OK=1)
      DECRYPT BLOB($vbEncrypted;$vbPublicKey)
      BLOB TO VARIABLE($vbEncrypted;$vtDecrypted)
      CREATE RECORD([Private])
      [Private]Info:=$vtDecrypted
      SAVE RECORD([Private])
   End if
End if
```

- **Using keypairs**

A company wants to use the Internet to exchange information. Each subsidiary receives private information and also sends information to the corporate office. Consequently there are two requirements:
- The recipient only should be able to read the message,
- The recipient must have proof that the message was sent by the sender himself.

1) The corporate office and each subsidiary generate their own key pairs (with the ***GENERATE_KEYS_TXT*** method).

2) The private key is kept secret by both sides. Each subsidiary sends its public key to the corporate office who, in its turn, sends its public key too. This key transfer does not need to be done through a secured channel as the public key is not enough to decrypt the message.

3) To encrypt the information to send, the subsidiary or the corporate house executes the ***ENCRYPT_INFO_2*** method which uses the sender's private key and the recipient's public key to encrypt the information:

```
 `Method ENCRYPT_INFO_2
C_BLOB($vbEncrypted;$vbPrivateKey;$vbPublicKey)
C_TEXT($vtEncrypt)
C_TIME($vtDocRef)

$vtEncrypt:=[Private]Info
VARIABLE TO BLOB($vtEncrypt;$vbEncrypted)
 ` Your own private key is loaded...
DOCUMENT TO BLOB("PrivateKey.txt";$vbPrivateKey)
If(OK=1)
 ` ...and the recipient' s public key
   ALERT("Please select the recipient' s public key.")
   $vhDocRef:=Open document("") `Public key to load
   If(OK=1)
      CLOSE DOCUMENT($vtDocRef)
      DOCUMENT TO BLOB(Document;$vbPublicKey)
 `BLOB encryption with the two keys as parameters
      ENCRYPT BLOB($vbEncrypted;$vbPrivateKey;$vbPublicKey)
      BLOB TO DOCUMENT("Update.txt";$vbEncrypted)
   End if
End if
```

4) The encrypted file can then be sent to the recipient via the Internet. If a third person gets hold of it, he or she will not be able to decrypt the message, even if he or she has the public keys as the recipient's private key will also be required.

5) Each recipient can decrypt the document by using his/her own private key and the sender's public key:

```
 `Method DECRYPT_INFO_2
C_BLOB($vbEncrypted;$vbPublicKey;$vbPrivateKey)
C_TEXT($vtDecrypted)
C_TIME($vhDocRef)

ALERT("Please select the encrypted document.")
$vhDocRef:=Open document("") `Select the Update.txt file
If(OK=1)
   CLOSE DOCUMENT($vhDocRef)
   DOCUMENT TO BLOB(Document;$vbEncrypted)
 `Your own private key is loaded
   DOCUMENT TO BLOB("PrivateKey.txt";$vbPrivateKey)
   If(OK=1)
 ` ...and the sender' s public key
      ALERT("Please select the sender' s public key.")
      $vhDocRef:=Open document("") `Public key to load
      If(OK=1)
         CLOSE DOCUMENT($vhDocRef)
         DOCUMENT TO BLOB(Document;$vbPublicKey)
 `Decrypting the BLOB with two keys as parameters
         DECRYPT BLOB($vbEncrypted;$vbPublicKey;$vbPrivateKey)
         BLOB TO VARIABLE($vbEncrypted;$vtDecrypted)
         CREATE RECORD([Private])
         [Private]Info:=$vtDecrypted
         SAVE RECORD([Private])
      End if
   End if
End if
```

# EXPAND BLOB

EXPAND BLOB ( blob )

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ➡ | BLOB to expand |

## Description

The **EXPAND BLOB** command expands the BLOB *blob* that was previously compressed using the **COMPRESS BLOB** command.

After the call, the OK variable is set to 1 if the BLOB has been expanded. If the expansion could not be performed, the OK variable is set to 0.

If the expansion could not be performed because of a lack of memory, no error is generated and the method resumes its execution.
In any other case (i.e. the BLOB has not been compressed or is damaged), the error -10600 is generated. This error can be trapped using the **ON ERR CALL** command.

To check if a BLOB has been compressed, use the **BLOB PROPERTIES** command.

## Example 1

This example tests if the BLOB *vxMyBlob* is compressed and, if so, expands it:

```
BLOB PROPERTIES(vxMyBlob;$vlCompressed;$vlExpandedSize;$vlCurrentSize)
If($vlCompressed#Is not compressed)
   EXPAND BLOB(vxMyBlob)
End if
```

## Example 2

This example allows you to select a document and then expand it, if it is compressed:

```
$vhDocRef :=Open document("")
If(OK=1)
   CLOSE DOCUMENT($vhDocRef)
   DOCUMENT TO BLOB(Document;vxBlob)
   If(OK=1)
      BLOB PROPERTIES(vxBlob;$vlCompressed;$vlExpandedSize;$vlCurrentSize)
      If($vlCompressed#Is not compressed)
         EXPAND BLOB(vxBlob)
         If(OK=1)
            BLOB TO DOCUMENT(Document;vxBlob)
         End if
      End if
   End if
End if
```

## System variables and sets

The OK variable is set to 1 if the BLOB has been successfully expanded, otherwise it is set to 0.

# INSERT IN BLOB

INSERT IN BLOB ( blob ; offset ; len {; filler} )

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ⇒ | BLOB into which bytes will be inserted |
| offset | Longint | ⇒ | Starting position where bytes will be inserted |
| len | Longint | ⇒ | Number of bytes to be inserted |
| filler | Longint | ⇒ | Default byte value (0x00..0xFF) 0x00 if omitted |

## Description

The **INSERT IN BLOB** command inserts the number of bytes specified by *len* into the BLOB *blob* at the position specified by *offset*. The BLOB then becomes *len* bytes larger.

If you do not specify the optional *filler* parameter, the bytes inserted into the BLOB are set to *0x00*. Otherwise, the bytes are set to the value you pass in *filler* (modulo 256 — 0..255).

Before the call, you pass in the *offset* parameter the position of the insertion relative to the beginning of the BLOB.

# INTEGER TO BLOB

INTEGER TO BLOB ( entier ; blob ; ordreOctet {; offset | *} )

| Parameter | Type | | Description |
|---|---|---|---|
| entier | Longint | ⇒ | Integer value to write into the BLOB |
| blob | BLOB | ⇒ | BLOB to receive the Integer value |
| ordreOctet | Longint | ⇒ | 0 Native byte ordering 1 Macintosh byte ordering 2 PC byte ordering |
| offset \| * | Variable, Operator | ⇒ | Offset expressed in bytes within the BLOB or * to append the value |
| | | ⇐ | New offset after writing if not * |

## Description

The **INTEGER TO BLOB** command writes the 2-byte Integer value *integer* into the BLOB *blob*.

The *byteOrder* parameter fixes the byte ordering of the 2-byte Integer value to be written. You pass one of the following predefined constants provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Macintosh byte ordering | Longint | 1 |
| Native byte ordering | Longint | 0 |
| PC byte ordering | Longint | 2 |

**Note regarding Platform Independence:** If you exchange BLOBs between the Macintosh and PC platforms, it is up to you to manage byte swapping issues when using this command.

If you specify the * optional parameter, the 2-byte Integer value is appended to the BLOB and the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of Integer, Long Integer, Real or Text values (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter or the *offset* variable parameter, the 2-byte Integer value is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the *offset* variable parameter, the 2-byte Integer value is written at the byte offset (starting from zero) within the BLOB. No matter where you write the 2-byte Integer value, the size of the BLOB is increased according to the location you passed (plus up to 2 bytes, if necessary). Newly allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the *offset* variable parameter is returned, incremented by the number of bytes that have been written. Therefore, you can reuse that same variable with another BLOB writing command to write another value.

## Example 1

After executing this code:

```
INTEGER TO BLOB(0x0206;vxBlob;Native byte ordering)
```

- The size of *vxBlob* is 2 bytes
- On PowerPC platform: *vxBLOB{0}* = *$02* and *vxBLOB{1}* = *$06*
- On Intel platform: *vxBLOB{0}* = *$06* and *vxBLOB{1}* = *$02*

## Example 2

After executing this code:

```
INTEGER TO BLOB(0x0206;vxBlob;Macintosh byte ordering)
```

- The size of *vxBlob* is 2 bytes
- On all platforms *vxBLOB{0}* = *$02* and *vxBLOB{1}* = *$06*

## Example 3

After executing this code:

```
INTEGER TO BLOB(0x0206;vxBlob;PC byte ordering)
```

- The size of *vxBlob* is 2 bytes
- On all platforms *vxBLOB{0}* = *$06* and *vxBLOB{1}* = *$02*

## Example 4

After executing this code:

```
SET BLOB SIZE(vxBlob;100)
INTEGER TO BLOB(0x0206;vxBlob;PC byte ordering;*)
```

- The size of *vxBlob* is 102 bytes
- On all platforms *vxBLOB{100}* = *$06* and *vxBLOB{101}* = *$02*
- The other bytes of the BLOB are left unchanged

## Example 5

After executing this code:

```
SET BLOB SIZE(vxBlob;100)
vlOffset:=50
INTEGER TO BLOB(518;vxBlob;Macintosh byte ordering;vlOffset)
```

- The size of *vxBlob* is 100 bytes
- On all platforms *vxBLOB{50}* = *$02* and *vxBLOB{51}* = *$06*
- The other bytes of the BLOB are left unchanged
- The variable *vlOffset* has been incremented by 2 (and is now equal to 52)

# LIST TO BLOB

LIST TO BLOB ( list ; blob {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ⇒ | Hierarchical list to store in the BLOB |
| blob | BLOB | ⇒ | BLOB to receive the Hierarchical list |
| * | Operator | ⇒ | * to append the value |

## Description

The **LIST TO BLOB** command stores the hierarchical list *list* in the BLOB *blob*.

If you specify the * optional parameter, the hierarchical list is appended to the BLOB and the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of variables or lists (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter, the hierarchical list is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

Wherever the hierarchical list is stored, the size of the BLOB will be increased if necessary according to the specified location (plus up to the size of the list if necessary). Modified bytes (other than the ones you set) are reset to 0 (zero).

**WARNING:** If you use a BLOB for storing lists, you must later use the command **BLOB to list** for reading back the contents of the BLOB, because lists are stored in BLOBs using a 4D internal format.

After the call, if the list has been successfully stored, the OK variable is set to 1. If the operation could not be performed, the OK variable is set to 0; for example, if there was not enough memory.

**Note regarding Platform Independence: LIST TO BLOB** and **BLOB to list** use a 4D internal format for handling lists stored in BLOBs. As a benefit, you do not need to worry about byte swapping between platforms when using these two commands. In other words, a BLOB created on Windows using those commands can be reused on Macintosh, and vice-versa.

## Example

See example for the command **BLOB to list**.

# LONGINT TO BLOB

LONGINT TO BLOB ( longInt ; blob ; byteOrder {; offset | *} )

| Parameter | Type | | Description |
|---|---|---|---|
| longInt | Longint | ➡ | Long Integer value to write into the BLOB |
| blob | BLOB | ➡ | BLOB to receive the Long Integer value |
| byteOrder | Longint | ➡ | 0 Native byte ordering 1 Macintosh byte ordering 2 PC byte ordering |
| offset \| * | Variable, Operator | ➡ | Offset within the BLOB (expressed in bytes) or * to append the value |
| | | ⬅ | New offset after writing if not * |

## Description

The **LONGINT TO BLOB** command writes the 4-byte Long Integer value *integer* into the BLOB *blob*.

The *byteOrder* parameter fixes the byte ordering of the 4-byte Long Integer value to be written. You pass one of the following predefined constants provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Macintosh byte ordering | Longint | 1 |
| Native byte ordering | Longint | 0 |
| PC byte ordering | Longint | 2 |

**Note regarding Platform Independence:** If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage byte swapping issues while using this command.

If you specify the * optional parameter, the 4-byte Long Integer value is appended to the BLOB and the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of Integer, Long Integer, Real or Text values (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter nor the *offset* variable parameter, the 4-byte Long Integer value is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the *offset* variable parameter, the 4-byte Long Integer value is written at the offset (starting from zero) within the BLOB. No matter where you write the 4-byte Long Integer value, the size of the BLOB is increased according to the location you passed (plus up to 4 bytes, if necessary). New allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the *offset* variable parameter is returned, incremented by the number of bytes that have been written. Therefore, you can reuse that same variable with another BLOB writing command to write another value.

## Example 1

After executing this code:

```
LONGINT TO BLOB(0x01020304;vxBlob;Native byte ordering)
```

- The size of *vxBlob* is 4 bytes
- On PowerPC platform: *vxBLOB{0}=$01*, *vxBLOB{1}=$02*, *vxBLOB{2}=$03*, *vxBLOB{3}=$04*
- On Intel platform: *vxBLOB{0}=$04*, *vxBLOB{1}=$03*, *vxBLOB{2}=$02*, *vxBLOB{3}=$01*

## Example 2

After executing this code:

```
LONGINT TO BLOB(0x01020304;vxBlob;Macintosh byte ordering)
```

- The size of *vxBlob* is 4 bytes
- On all platforms *vxBLOB{0}=$01*, *vxBLOB{1}=$02*, *vxBLOB{2}=$03*, *vxBLOB{3}=$04*

## Example 3

After executing this code:

```
LONGINT TO BLOB(0x01020304;vxBlob;PC byte ordering)
```

- The size of *vxBlob* is 4 bytes
- On all platforms *vxBLOB{0}=$04*, *vxBLOB{1}=$03*, *vxBLOB{2}=$02*, *vxBLOB{3}=$01*

## Example 4

After executing this code:

```
SET BLOB SIZE(vxBlob;100)
LONGINT TO BLOB(0x01020304;vxBlob;PC byte ordering;*)
```

- The size of *vxBlob* is 104 bytes
- On all platforms *vxBLOB{100}=$04*, *vxBLOB{101}=$03*, *vxBLOB{102}=$02*, *vxBLOB{103}=$01*
- The other bytes of the BLOB are left unchanged

## Example 5

After executing this code:

```
SET BLOB SIZE(vxBlob;100)
vlOffset:=50
LONGINT TO BLOB(0x01020304;vxBlob;Macintosh byte ordering;vlOffset)
```

- The size of *vxBlob* is 100 bytes
- On all platforms *vxBLOB{50}=$01*, *vxBLOB{51}=$02*, *vxBLOB{52}=$03*, *vxBLOB{53}=$04*
- The other bytes of the BLOB are left unchanged
- The variable *vlOffset* has been incremented by 4 (and is now equal to 54)

# REAL TO BLOB

REAL TO BLOB ( real ; blob ; realFormat {; offset | *} )

| Parameter | Type | | Description |
|---|---|---|---|
| real | Real | ➡ | Real value to write into the BLOB |
| blob | BLOB | ➡ | BLOB to receive the Real value |
| realFormat | Longint | ➡ | 0 Native real format 1 Extended real format 2 Macintosh Double real format 3 Windows Double real format |
| offset \| * | Variable, Operator | ➡ | Offset within the BLOB (expressed in bytes) or * to append the value |
| | | ⬅ | New offset after writing if not * |

## Description

The **REAL TO BLOB** command writes the Real value *real* into the BLOB *blob*.

The *realFormat* parameter fixes the internal format and byte ordering of the Real value to be written. You pass one of the following predefined constants provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Extended real format | Longint | 1 |
| Macintosh double real format | Longint | 2 |
| Native real format | Longint | 0 |
| PC double real format | Longint | 3 |

**Platform Independence Note:** If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage real formats and byte swapping issues when using this command.

If you specify the * optional parameter, the Real value is appended to the BLOB; the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of Integer, Long Integer, Real or Text values (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter or the *offset* variable parameter, the Real value is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the *offset* variable parameter, the Real value is written at the offset (starting from zero) within the BLOB. No matter where you write the Real value, the size of the BLOB is increased according to the location you passed (plus up to 8 or 10 bytes, if necessary). New allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the *offset* variable parameter is returned, incremented by the number of bytes that have been written. Therefore, you can reuse that same variable with another BLOB writing command to write another value.

## Example 1

After executing this code:

```
C_REAL(vrValue)
vrValue:=...
REAL TO BLOB(vrValue;vxBlob;Native real format)
```

- On all platforms, the size of *vxBlob* is 8 bytes

## Example 2

After executing this code:

```
C_REAL(vrValue)
vrValue:=...
REAL TO BLOB(vrValue;vxBlob;Extended real format)
```

- On all platforms, the size of *vxBlob* is 10 bytes

## Example 3

After executing this code:

```
C_REAL(vrValue)
vrValue:=...
REAL TO BLOB(vrValue;vxBlob;Macintosh double real format) ` or Windows double real format
```

- On all platforms, the size of *vxBlob* is 8 bytes

## Example 4

After executing this code:

```
SET BLOB SIZE(vxBlob;100)
C_REAL(vrValue)
vrValue:=...
INTEGER TO BLOB(vrValue;vxBlob;PC double real format) ` or Macintosh double real format
```

- On all platforms, the size of *vxBlob* is 8 bytes

## Example 5

After executing this code:

```
SET BLOB SIZE(vxBlob;100)
REAL TO BLOB(vrValue;vxBlob;Extended real format;*)
```

- On all platforms, the size of *vxBlob* is 110 bytes
- On all platforms, the real value is stored at the bytes #100 to #109
- The other bytes of the BLOB are left unchanged

## Example 6

After executing this code:

```
SET BLOB SIZE(vxBlob;100)
C_REAL(vrValue)
vrValue:=...
vlOffset:=50
REAL TO BLOB(vrValue;vxBlob;PC double real format;vlOffset) ` or Macintosh double real format
```

- On all platforms, the size of *vxBlob* is 100 bytes
- On all platforms, the real value is stored in the bytes #50 to #57
- The other bytes of the BLOB are left unchanged
- The variable *vlOffset* has been incremented by 8 (and is now equal to 58)

# SET BLOB SIZE

SET BLOB SIZE ( blob ; size {; filler} )

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ➡ | BLOB field or variable |
| size | Longint | ➡ | New size of the BLOB |
| filler | Longint | ➡ | ASCII code of filler character |

## Description

**SET BLOB SIZE** resizes the BLOB *blob* according to the value passed in *size*.

If you want to allocate new bytes to a BLOB and want to have those bytes initialized to a specific value, pass the value (0..255) into the *filler* optional parameter.

## Example 1

When you are through with a large process or interprocess BLOB, it is good idea to free the memory it occupies. To do so, write:

```
SET BLOB SIZE(aProcessBLOB;0)
SET BLOB SIZE(◊anInterprocessBLOB;0)
```

## Example 2

The following example creates a BLOB of 16K filled of 0xFF:

```
C_BLOB(vxData)
SET BLOB SIZE(vxData;16*1024;0xFF)
```

## Error Handling

If you cannot resize a BLOB due to insufficient memory, the error -108 is generated. You can trap this error using an **ON ERR CALL** interruption method.

# TEXT TO BLOB

TEXT TO BLOB ( text ; blob {; textFormat {; offset | *}} )

| Parameter | Type | | Description |
|---|---|---|---|
| text | String | ➡ | Text to write into the BLOB |
| blob | BLOB | ➡ | BLOB to receive the text |
| textFormat | Longint | ➡ | Format and character set of text |
| offset \| * | Variable, Operator | ➡ | Offset within the BLOB (expressed in bytes) or * to append the value |
| | | ⬅ | New offset after writing if not * |

## Description

The **TEXT TO BLOB** command writes the Text value *text* into the BLOB *blob*.

The *textFormat* parameter can be used to set the internal format and the character set of the text value to be written. To do this, pass one of the following constants (found in the "**BLOB**" theme) in the *textFormat* parameter:

| Constant | Type | Value |
|---|---|---|
| Mac C string | Longint | 0 |
| Mac Pascal string | Longint | 1 |
| Mac text with length | Longint | 2 |
| Mac text without length | Longint | 3 |
| UTF8 C string | Longint | 4 |
| UTF8 text with length | Longint | 5 |
| UTF8 text without length | Longint | 6 |

If you omit the *textFormat* parameter, by default 4D uses the Mac C string format. In databases created beginning with version 11, 4D works by default with the Unicode character set (UTF8) for managing text, so it is recommended to use this character set.

**Notes:**

- The "UTF8" constants can only be used when the application runs in Unicode mode.
- The "Mac" constants cannot work with texts greater than 32 KB.
- If you want to work with character sets other than UTF8, use the **CONVERT FROM TEXT** command.

The following table describes each of these formats:

| Text format | Description and Examples | |
|---|---|---|
| C string | The text is ended by a NULL character (ASCII code $00). | |
| | *UTF8* | "" --> $00 |
| | | "Café" --> $43 61 66 C3 A9 00 |
| | *Mac* | "" --> $00 |
| | | "Café" --> $43 61 66 8E 00 |
| Pascal string | The text is preceded by a 1-byte length. | |
| | *UTF8* | - |
| | | - |
| | *Mac* | "" --> $00 |
| | | "Café" --> $04 43 61 66 8E |
| Text with length | The text is preceded by a 4-byte (UTF8) or 2-byte (Mac) length. | |
| | *UTF8* | "" --> $00 00 00 00 |
| | | "Café" --> $00 00 00 05 43 61 66 C3 A9 |
| | *Mac* | "" --> $00 00 |
| | | "Café" --> $00 04 43 61 66 8E |
| Text without length | The text is composed only of its characters. | |
| | *UTF8* | "" --> No data |
| | | "Café" --> $43 61 66 C3 A9 |
| | *Mac* | "" --> No data |
| | | "Café" --> $43 61 66 8E |

If you specify the * optional parameter, the Text value is appended to the BLOB; the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of Integer, Long Integer, Real or Text values (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter nor the *offset* variable parameter, the Text value is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the *offset* variable parameter, the Text value is written at the offset (starting from zero) within the BLOB. No matter where you write the Text value, the size of the BLOB is, increased according to the location you passed (plus up to the size of the text, if necessary). New allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the *offset* variable parameter is returned, incremented by the number of bytes that have been written. Therfore, you can reuse that same variable with another BLOB writing command to write another value.

## Example

After executing this code:

```
SET BLOB SIZE(vxBlob;0)
C_TEXT(vtValue)
vtValue:="Café" ` Length of vtValue is 4 bytes
TEXT TO BLOB(vtValue;vxBlob;Mac C string) ` Size of BLOB becomes 5 bytes
TEXT TO BLOB(vtValue;vxBlob;Mac Pascal string) ` Size of BLOB becomes 5 bytes
TEXT TO BLOB(vtValue;vxBlob;Mac text with length) ` Size of BLOB becomes 6 bytes
TEXT TO BLOB(vtValue;vxBlob;Mac text without length) ` Size of BLOB becomes 4 bytes
TEXT TO BLOB(vtValue;vxBlob;UTF8 C string) ` Size of BLOB becomes 6 bytes
TEXT TO BLOB(vtValue;vxBlob;UTF8 text with length) ` Size of BLOB becomes 9 bytes
TEXT TO BLOB(vtValue;vxBlob;UTF8 text without length) ` Size of BLOB becomes 5 bytes
```

# VARIABLE TO BLOB

VARIABLE TO BLOB ( variable ; blob {; offset | *} )

| Parameter | Type | | Description |
|---|---|---|---|
| variable | Variable | ➡ | Variable to store in the BLOB |
| blob | BLOB | ➡ | BLOB to receive the variable |
| offset \| * | Variable, Operator | ➡ | Offset within the BLOB (expressed in bytes) or * to append the value |
| | | ⬅ | New offset after writing if not * |

## Description

The **VARIABLE TO BLOB** command stores the variable *variable* in the BLOB *blob*.

If you specify the * optional parameter, the variable is appended to the BLOB and the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of variables or lists (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter or the *offset* variable parameter, the variable is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the *offset* variable parameter, the variable is written at the offset (starting from zero) within the BLOB. No matter where you write the variable, the size of the BLOB is increased according to the location you passed (plus the size of the variable, if necessary). Newly allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the *offset* variable parameter is returned, incremented by the number of bytes that have been written. Therefore, you can reuse that same variable with another BLOB writing command to write another variable or list.

**VARIABLE TO BLOB** accepts any type of variable (including other BLOBs), except the following:

- Pointer
- Array of pointers

Note that:

- if you store a Long Integer variable that is a reference to a hierarchical list (ListRef), **VARIABLE TO BLOB** stores the Long Integer variable, not the list. To store and retrieve hierarchical lists in and from a BLOB, use the **LIST TO BLOB** and **BLOB to list** commands.
- if you pass a **C_OBJECT** object in the *variable* parameter, the command places it in the BLOB as JSON in UTF-8. If the object contains pointers, their dereferenced values are stored in the BLOB, not the pointers themselves.

**WARNING:** If you use a BLOB for storing variables, you must later use the command **BLOB TO VARIABLE** for reading back the contents of the BLOB, because variables are stored in BLOBs using a 4D internal format.

After the call, if the variable has been successfully stored, the OK variable is set to 1. If the operation could not be performed, the OK variable is set to 0; for example, there was not enough memory.

**Note regarding Platform Independence: VARIABLE TO BLOB** and **BLOB TO VARIABLE** use a 4D internal format for handling variables stored in BLOBs. As a benefit, you do not need to worry about byte swapping between platforms while using these two commands. In other words, a BLOB created on Windows using either of these commands can be reused on Macintosh, and vice-versa.

## Example 1

The two following project methods allow you to quickly store and retrieve arrays into and from documents on disk:

```
  ` SAVE ARRAY project method
  ` SAVE ARRAY ( String ; Pointer )
  ` SAVE ARRAY ( Document ; -> Array )
C_STRING(255;$1)
C_POINTER($2)
C_BLOB($vxArrayData)
VARIABLE TO BLOB($2->;$vxArrayData) ` Store the array into the BLOB
COMPRESS BLOB($vxArrayData) ` Compress the BLOB
BLOB TO DOCUMENT($1;$vxArrayData) ` Save the BLOB on disk
```

```
   ` LOAD ARRAY project method
   ` LOAD ARRAY ( String ; Pointer )
   ` LOAD ARRAY ( Document ; -> Array )
C_STRING(255;$1)
C_POINTER($2)
C_BLOB($vxArrayData)
DOCUMENT TO BLOB($1;$vxArrayData) ` Load the BLOB from the disk
EXPAND BLOB($vxArrayData) ` Expand the BLOB
BLOB TO VARIABLE($vxArrayData;$2->) ` Retrieve the array from the BLOB
```

After these methods have been added to your application, you can write:

```
ARRAY STRING(...;asAnyArray;...)
` ...
SAVE ARRAY($vsDocName;->asAnyArray)
` ...
LOAD ARRAY($vsDocName;->asAnyArray)
```

## Example 2

The two following project methods allow you to quickly store and retrieve any set of variables into and from a BLOB:

```
 //STORE VARIABLES INTO BLOB project method
 //STORE VARIABLES INTO BLOB ( Pointer { ; Pointer ... { ; Pointer } } )
 //STORE VARIABLES INTO BLOB ( BLOB { ; Var1 ... { ; Var2 } } )
C_POINTER($1)
C_LONGINT($vlParam)

SET BLOB SIZE($1->;0)
For($vlParam;2;Count parameters)
   VARIABLE TO BLOB($1->;$1->;*)
End for


 //RETRIEVE VARIABLES FROM BLOB project method
 //RETRIEVE VARIABLES FROM BLOB ( Pointer { ; Pointer ... { ; Pointer } } )
 //RETRIEVE VARIABLES FROM BLOB ( BLOB { ; Var1 ... { ; Var2 } } )
C_POINTER($1)
C_LONGINT($vlParam;$vlOffset)

$vlOffset:=0
For($vlParam;2;Count parameters)
   BLOB TO VARIABLE($1->;$1->;$vlOffset)
End for
```

After these methods have been added to your application, you can write:

```
STORE VARIABLES INTO BLOB(->vxBLOB;->vgPicture;->asAnArray;->alAnotherArray)
 // ...
RETRIEVE VARIABLES FROM BLOB(->vxBLOB;->vgPicture;->asAnArray;->alAnotherArray)
```

## System variables and sets

The OK variable is set to 1 if the variable has been successfully stored, otherwise it is set to 0.

# Boolean

- Boolean Commands
- False
- Not
- True

## Boolean Commands

4D includes Boolean functions, are used for Boolean calculations:

```
True
False
Not
```

### Example

This example sets a *Boolean* variable based on the value of a button. It returns True in *myBoolean* if the *myButton* button was clicked and False if the button was not clicked. When a button is clicked, the button variable is set to 1.

```
If(myButton=1) ` If the button was clicked
   myBoolean:=True ` myBoolean is set to True
Else ` If the button was not clicked,
   myBoolean:=False ` myBoolean is set to False
End if
```

The previous example can be simplified into one line.

```
myBoolean:=(myButton=1)
```

# False

False -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | False |

## Description

**False** returns the Boolean value False.

## Example

The following example sets the variable *vbOptions* to False:

```
vbOptions:=False
```

# Not

Not ( boolean ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| boolean | Boolean | ⇒ | Boolean value to negate |
| Function result | Boolean | ⇒ | Opposite of Boolean |

## Description

The **Not** function returns the negation of *boolean*, changing True to False or False to True.

## Example

This example first assigns True to a variable, then changes the variable value to False, and then back to True.

```
vResult:=True ` vResult is set to True
vResult:=Not(vResult) ` vResult is set to False
vResult:=Not(vResult) ` vResult is set to True
```

# True

True -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | True |

## Description

**True** returns the Boolean value True.

## Example

The following example sets the variable *vbOptions* to True:

```
vbOptions:=True
```

# Cache Management

4D integrates a built-in data cache scheme for accelerating I/O operations. The fact that data modifications are, for some time, present in the data cache and not on the disk is transparent to your coding. For example, if you issue a **QUERY** call, the 4D database engine integrates the data cache in the query operation.

The database cache manager has been fully rewritten in 4D v16, allowing it to take advantage of 64-bit environments. Automatically enabled and optimized, it can however be dynamically configured or analyzed with the commands from this theme.

Note also that a Cache flush periodicity selector can be used with the **SET DATABASE PARAMETER** and **Get database parameter** commands.

- Cache info New 16.0
- FLUSH CACHE Updated 16.0
- Get cache size New 16.0
- GET MEMORY STATISTICS Updated 16.0
- SET CACHE SIZE New 16.0

# Cache info

Cache info {{ dbFilter )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| dbFilter | Object | ⇒ | Defines list of attributes to be returned (filtered per DB) |
| Function result | Object | ⇐ | Information about cache |

## 64-bit only

This command only works in 4D 64-bit versions.

## Description

The **Cache info** command returns an object that contains detailed information about the current cache contents (used memory, loaded tables and indexes, etc.)

**Note:** This command only works in local mode (4D Server and 4D); it must not be used from 4D in remote mode.

By default, returned information refers to the running database only. The optional *dbFilter* object parameter allows you to specify the scope of the command:

- pass the "dbFilter" attribute with the "All" value to get cache information about all running databases, including components.
- pass the "dbFilter" attribute with a "" (empty string) value to get information about the current database only (equivalent to omitting the *dbFilter* parameter).

The **Cache info** command returns a single object that contains all the relevant information about the cache. The returned object has the following basic structure:

```
{
   "maxMem": Maximum cache size (real),
   "usedMem": Current cache size (real),
   "objects": [...] Array of objects currently loaded in cache
}
```

Elements of the *objects* array are root objects (tables, indexes, Blobs, etc.) which are currently loaded in the cache. Each element contains specific attributes that describe its current status. For more information about advanced interpretation of this data, please contact your local Technical Service department.

## Example

You want to get cache information for the current database:

```
C_OBJECT($cache)
$cache:=Cache info
```

You want to get cache information for the database and all opened components:

```
C_OBJECT($dbFilter)
OB SET($dbFilter;"dbFilter";"All")
$cache:=Cache info($dbFilter)
```

# FLUSH CACHE

FLUSH CACHE {( size | * )}

| Parameter | Type | | Description |
|---|---|---|---|
| size \| * | Real, Operator | ⇒ | * to completely free cache memory, or number of bytes to free in cache |

## Description

The **FLUSH CACHE** command immediately saves the data buffers to disk. All changes that have been made to the database are stored on disk.

By default, the current cache memory is left untouched, which means that its data continues to be used for subsequent reading accesses. Optionally, you can pass a parameter to modify its contents:

- pass * to save the cache and free up entire cache memory.
- pass a *size* value to save the cache and free up only the *size* number of bytes from the cache.

**Note:** Passing a parameter to this command is reserved for testing purposes. For performance reasons, it is not recommended to free up the cache in the production environment.

In normal cases, you should not call this command, as 4D saves data modifications on a regular basis. The **Flush Cache every X Seconds (Minutes)** option on the **Database/Memory page** of the Database Settings, which specifies how often to save, is typically used to control cache flushing. We recommend using the default value of 20 seconds. Note also that the Cache flush periodicity parameter can be set and read using the **SET DATABASE PARAMETER** and **Get database parameter** commands.

# Get cache size

Get cache size -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Real |  | Size of database cache in bytes |

## Description

The **Get cache size** command returns the current database cache size in bytes.

**Note:** This command only works in local mode (4D Server and 4D); it must not be used from 4D in remote mode.

## Example

See example for **SET CACHE SIZE** command.

# GET MEMORY STATISTICS

GET MEMORY STATISTICS ( infoType ; arrNames ; arrValues ; arrCount )

| Parameter | Type | | Description |
|---|---|---|---|
| infoType | Longint | ⇨ | Selector of information to get |
| arrNames | Text array | ⇦ | Information titles |
| arrValues | Real array | ⇦ | Information values |
| arrCount | Real array | ⇦ | Number of objects concerned (if available) |

## Description

The **GET MEMORY STATISTICS** command recovers information related to the use of the data cache by 4D. This information can be used to analyze the functioning of the application.

In the *infoType* parameter, pass a value indicating the type of information that you want to obtain:

- 1 = General memory information. This information is also available via the Runtime Explorer: size of physical, virtual, free and used memory, stack memory and free stack memory, etc.
- 2 = Summary of database cache occupancy statistics (4D 32-bit only).

**Compatibility Note:** Value 2 is deprecated for 4D 64-bit versions. For these versions, we recommend using the **Cache info** command to get cache statistics.

After the command has been executed, the statistics are provided in the *arrNames*, *arrValues* and *arrCount* arrays. For more information about advanced interpretation of this data, please contact your local Technical Service department.

# SET CACHE SIZE

SET CACHE SIZE ( size {; minFreeSize} )

| Parameter | Type | | Description |
|---|---|---|---|
| size | Real | ⇒ | Size of database cache in bytes |
| minFreeSize | Real | ⇒ | Minimum number of bytes to release when cache is full |

## 64-bit only

This command only works in 4D 64-bit versions.

## Description

The **SET CACHE SIZE** command sets the database cache size dynamically and, optionally, sets the mininum byte size at which to start to free memory.

**Note:** This command only works in local mode (4D Server and 4D); it cannot be used in 4D remote mode.

In *size*, pass the new size for the database cache in bytes. This new size is applied dynamically when the command is executed.

In *minFreeSize*, pass the minimum size of memory to release from the database cache when the engine needs to make space in order to allocate an object to it (value in bytes). The purpose of this option is to reduce the number of times that data is released from the cache in order to obtain better performance.
By default, if this option is not used, 4D unloads at least 10% of the cache when space is needed. If your database works with a large cache, it could be advantageous to use a fixed size that does not depend on the cache size. You can adjust this setting according to the size of the blocks of data being handled in your database.

## Example

You want to add 100 MB to the current database cache size. You can write:

```
C_REAL($currentCache)
$currentCache:=Get cache size
 // current cache size is, for example, 419430400
SET CACHE SIZE($currentCache+10000000)
 // current cache size is now 519430400
```

# Communications

- GET SERIAL PORT MAPPING
- RECEIVE BUFFER
- RECEIVE PACKET
- RECEIVE RECORD
- RECEIVE VARIABLE
- SEND PACKET
- SEND RECORD
- SEND VARIABLE
- SET CHANNEL
- SET TIMEOUT
- USE CHARACTER SET

# GET SERIAL PORT MAPPING

GET SERIAL PORT MAPPING ( numArray ; nameArray )

| Parameter | Type | | Description |
|---|---|---|---|
| numArray | Longint array | ⇐ | Array of port numbers |
| nameArray | String array | ⇐ | Array of port names |

## Description

The **GET SERIAL PORT MAPPING** command returns two arrays, *numArray* and *nameArray*, containing the serial port numbers and the serial port names of the current machine.

This command is useful under Mac OS X, where the operating system dynamically allocates the port number when using a USB serial adapter. You can address any extended serial port using its name (static), regardless of its actual number.

**Note:** This command does not return meaningful values with standard ports. If you want to address a standard port, you must pass its value (0 or 1) directly using the **SET CHANNEL** command (former operation of 4D).

## Example

This project method can be used to address the same serial port (without protocol), regardless of the number that has been assigned to it:

```
ARRAY TEXT($arrPortNames;0)
ARRAY LONGINT($arrPortNums;0)
C_LONGINT($vPortNum;$vFinalPortNum)

 `Find out the current numbers of the serial ports
GET SERIAL PORT MAPPING($arrPortNums;$arrPortNames)
$vPortNum:=Find in array($arrPortNames;vPortName)
 ` vPortName contains the name of the port to be used: it may come from a dialog box,
 ` a value stored in a field, etc.
If(arrPortNums{$vPortNum}=0)
   $vFinalPortNum:=0 `special case under Mac OS X
Else
   $vFinalPortNum:=arrPortNums{$vPortNum}+100
End if
SET CHANNEL($vFinalPortNum;params) `params contains the communication parameters
... `Carry out the desired operations
SET CHANNEL(11) `Closing of port
```

# RECEIVE BUFFER

RECEIVE BUFFER ( receiveVar )

| Parameter | Type | | Description |
|---|---|---|---|
| receiveVar | Text variable | ➡ | Variable to receive data |

## Description

**RECEIVE BUFFER** reads the serial port that was previously opened with **SET CHANNEL**. The serial port has a buffer that fills with characters until a command reads from the buffer. **RECEIVE BUFFER** gets the characters from the serial buffer, put them into *receiveVar* then clears the buffer. If there are no characters in the buffer, then *receiveVar* will contain nothing.

**On Windows**

The Windows serial port buffer is limited in size to 10 Kbytes. This means that the buffer can overflow. When it is full and new characters are received, the new characters replace the oldest characters. The old characters are lost; therefore, it is essential that the buffer is read quickly when new characters are received.

**On Mac OS**

The Mac OS X serial port buffer capacity is, in theory, unlimited (depending on the available memory). If the buffer is full and new characters are received, the new characters replace the oldest characters. The old characters are lost; therefore, it is essential that the buffer is read quickly when new characters are received.

**RECEIVE BUFFER** is different from **RECEIVE PACKET** in that it takes whatever is in the buffer and then immediately returns. **RECEIVE PACKET** waits until it finds a specific character or until a given number of characters are in the buffer.

During the execution of **RECEIVE BUFFER**, the user can interrupt the reception by pressing **Ctrl-Alt-Shift** (Windows) or **Command-Option-Shift** (Macintosh). This interruption generates an error -9994 that you can catch with an error-handling method installed using **ON ERR CALL**.

## Example

The project method ***LISTEN TO SERIAL PORT*** uses **RECEIVE BUFFER** to get text from the serial port and accumulate it into a an interprocess variable:

```
  ` LISTEN TO SERIAL PORT
  ` Opening the serial port
SET CHANNEL(201;Speed 9600+Data bits 8+Stop bits one+Parity none)
<>IP_Listen_Serial_Port:=True
While(<>IP_Listen_Serial_Port)
   RECEIVE BUFFER($vtBuffer)
   If((Length($vtBuffer)+Length(<>vtBuffer))>MAXTEXTLEN)
      <>vtBuffer:=""
   End if
   <>vtBuffer:=<>vtBuffer+$Buffer
End while
```

At this point, any other process can read the interprocess *◊vtBuffer* to work with the data coming from the serial port.

To stop listening to the serial port, just execute:

```
  ` Stop listening to the serial port
◊IP_Listen_Serial_Port:=False
```

Note that access to the interprocess *◊vtBuffer* variable should be protected by a semaphore, so that processes will not conflict. See the command **Semaphore** for more information.

# RECEIVE PACKET

RECEIVE PACKET ( {docRef ;} receiveVar ; stopChar | numBytes )

| Parameter | Type | | Description |
|---|---|---|---|
| docRef | DocRef | ⇒ | Document reference number, or Current channel (serial port or document) |
| receiveVar | Text variable, BLOB variable | ⇒ | Variable to receive data |
| stopChar \| numBytes | String, Longint | ⇒ | Character(s) at which to stop receiving, or Number of bytes to receive |

## Description

**RECEIVE PACKET** reads characters from a serial port or from a document.

If *docRef* is specified, this command retrieves data from a document opened using **Open document**, **Create document** or **Append document**. If *docRef* is omitted, this command retrieves data from the serial port or the document opened using **SET CHANNEL**.

Whatever the source, the characters read are returned in *receiveVar*, which must be a Text, String or BLOB variable. If the characters have been sent by the **SEND PACKET** command, the type must correspond to that of the packet sent.

**Notes:**

- When the package received is of the BLOB type, the command does not take into account any character set defined by the **USE CHARACTER SET** command. The BLOB is returned without any modification.
- When the package received is of the Text type, the **RECEIVE PACKET** command supports Byte Order Marks (BOMs). In this case, if the current character set is of the Unicode type (UTF-8, UTF-16 or UTF-32), 4D attempts to identify a BOM among the first bytes received. If one is detected, it is filtered out of the *receiveVar* variable and 4D uses the character set that it defines instead of the current character set.

To read a particular number of characters, pass this number in *numBytes*. If the *receiveVar* variable is of the Text type, in a single call you can read up to 2 GB of text (theoretical value).

To receive data until a particular string (composed of one or more characters) is encountered, pass this string in *stopChar* (the string is not returned in *receiveVar*).

In this case, if the character string specified by *stopChar* is not found:

- When **RECEIVE PACKET** is reading a document, it will stop reading at the end of the document.
- When **RECEIVE PACKET** is reading from a serial port, it will attempt to wait indefinitely until the timeout (if any) has elapsed (see **SET TIMEOUT**) or until the user interrupts the reception (see below).

During execution of **RECEIVE PACKET**, the user can interrupt the reception by pressing **Ctrl-Alt-Shift** (Windows) or **Command-Option-Shift** (Macintosh). This interruption generates an error -9994 that you can catch with an error-handling method installed using **ON ERR CALL**. Usually, you will only have to handle interruption of a reception when communicating over a serial port.

When reading a document, the first **RECEIVE PACKET** begins reading at the beginning of the document. The reading of each subsequent packet begins at the character following the last byte read.

**Note:** This command is useful for document opened with **SET CHANNEL**. On the other hand, for a document opened with **Open document**, **Create document** or **Append document**, you can use the **Get document position** and **SET DOCUMENT POSITION** commands to get and change the location in the document where the next writing (**SEND PACKET**) or reading (**RECEIVE PACKET**) will occur.

When attempting to read past the end of a file, **RECEIVE PACKET** will return with the data read up to that point and the variable OK will be set to 1. Then, the next **RECEIVE PACKET** will return an empty string and set the OK variable to zero.

## Example 1

The following example reads 20 characters from a serial port into the variable *getTwenty*:

```
RECEIVE PACKET(getTwenty;20)
```

## Example 2

The following example reads data from the document referenced by the variable *myDoc* into the variable *vData*. It reads until it encounters a carriage return:

```
RECEIVE PACKET(myDoc;vData;Char(Carriage return))
```

## Example 3

The following example reads data from the document referenced by the variable *myDoc* into the variable *vData*. It reads until it encounters the *</TD>* (end of table cell) HTML tag:

```
RECEIVE PACKET(myDoc;vData;"</TD>")
```

## Example 4

The following example reads data from a document into fields. The data is stored as fixed-length fields. The method calls a subroutine to strip any trailing spaces (spaces at the end of the string). The subroutine follows the method:

```
$vhDocRef :=Open document("";"TEXT") ` Open a TEXT document
If(OK=1) ` If the document was opened
   Repeat ` Loop until no more data
      RECEIVE PACKET($vhDocRef;$Var1;15) ` Read 15 characters
      RECEIVE PACKET($vhDocRef;$Var2;15) ` Do same as above for second field
      If(($Var1#"")|($Var2#"")) ` If at least one of the fields is not empty
         CREATE RECORD([People]) ` Create a new record
         [People]First :=Strip($Var1) ` Save the first name
         [People]Last :=Strip($Var2) ` Save the last name
         SAVE RECORD([People]) ` Save the record
      End if
   Until(OK=0)
   CLOSE DOCUMENT($vhDocRef) ` Close the document
End if
```

The spaces at the end of the data are stripped by the following method, called **Strip**:

```
For($i;Length($1);1;-1) ` Loop from end of string to start
   If($1≤$i≥#" ") ` If it is not a space…
      $i :=-$i  ` Force the loop to end
   End if
End for
$0:=Delete string($1;-$i;Length($1)) ` Delete the spaces
```

## System variables and sets

After a call to **RECEIVE PACKET**, the OK system variable is set to 1 if the packet is received without error. Otherwise, the OK system variable is set to 0.

# RECEIVE RECORD

RECEIVE RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table into which to receive the record, or Default table, if omitted |

## Description

**RECEIVE RECORD** receives a record into *table* from the serial port or document opened by the **SET CHANNEL** command. The record must have been sent with **SEND RECORD**. When you execute **RECEIVE RECORD**, a new record is automatically created for *table*. If the record is received correctly, you must then use **SAVE RECORD** to save the new record.

The complete record is received. This means that pictures and BLOBs stored in or with the record are also received.

**Important:** When records are being sent and received using **SEND RECORD** and **RECEIVE RECORD**, the source table structure and the destination table structure must be compatible. If they are not, 4D will convert values according to the table definitions when **RECEIVE RECORD** is executed.

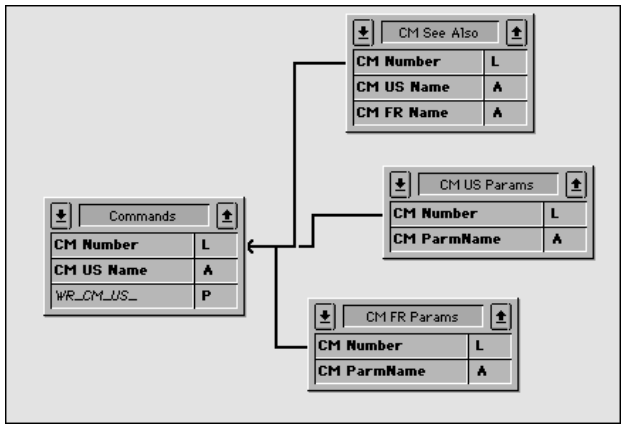
**Notes:**

1. If you receive a record from a document using this command, the document must have been opened using the **SET CHANNEL** command. You cannot use **RECEIVE RECORD** with a document opened with **Open document**, **Create document** or **Append document**.
2. During the execution of **RECEIVE RECORD**, the user can interrupt the reception by pressing **Ctrl-Alt-Shift** (Windows) or **Command-Option-Shift** (Macintosh). This interruption generates an error -9994 that you can catch with an error-handling method installed using **ON ERR CALL**. Usually, you only need to handle the interruption of a reception while communicating over a serial port.

## Example

A combined use of **SEND VARIABLE**, **SEND RECORD**, **RECEIVE VARIABLE** and **RECEIVE RECORD** is ideal for archiving data or for exchanging data between identical single-user databases used in different places. You can exchange data between 4D databases using the import/export commands such as **EXPORT TEXT** and **IMPORT TEXT**. However, if your data contains graphics and/or related tables, using **SEND RECORD** and **RECEIVE RECORD** is far more convenient.

For instance, consider a documentation system based on 4D and 4D Write. Since several writers in different locations wordwide work on it, we need a simple way to exchange data between the different databases. Here is a simplified view of the database structure:




The table *[Commands]* contains the description of each command or topic. The tables *[CM US Params]* and *[CM FR Params]* respectivily contain the parameter list for each command in English and in French. The table *[CM See Also]* contains the commands listed as reference (See Also section) for each command. Exchanging documentation between databases therefore consists in sending the *[Commands]* records and their related records. To do so, we use **SEND RECORD** and **RECEIVE RECORD**. In addition, we use **SEND VARIABLE** and **SEND RECORD** in order to mark the import/export document with tags.

Here is the (simplified) project method for exporting the documentation:

```
 // CM_EXPORT_SEL project method
 // This method works with the current selection of the [Commands] table

SET CHANNEL(12;"") // Let's the user create an open a channel document
If(OK=1)
 //  Tag the document with a variable that indicates its contents
 //  Note: the BUILD_LANG process variable indicates if US (English) or FR (French) data is sent
   $vsTag:="4DV6COMMAND"+BUILD_LANG
   SEND VARIABLE($vsTag)
 //  Send a variable indicationg how many [Commands] are sent
   $vlNbCmd:=Records in selection([Commands])
   SEND VARIABLE($vlNbCmd)
   FIRST RECORD([Commands])
 //  For each command
   For($vlCmd;1;$vlNbCmd)
 //  Send the [Commands] record
      SEND RECORD([Commands])
 //  Select all the related records
      RELATE MANY([Commands])
 //  Depending on the language, send a variable indicating
 //  the number of parameters that will follow
      Case of
         :(BUILD_LANG="US")
            $vlNbParm:=Records in selection([CM US Params])
         :(BUILD_LANG="FR")
            $vlNbParm:=Records in selection([CM FR Params])
      End case
      SEND VARIABLE($vlNbParm)
 //  Send the parameter records (if any)
      For($vlParm;1;$vlNbParm)
         Case of
            :(BUILD_LANG="US")
               SEND RECORD([CM US Params])
               NEXT RECORD([CM US Params])
            :(BUILD_LANG="FR")
               SEND RECORD([CM FR Params])
               NEXT RECORD([CM FR Params])
         End case
      End for
 //  Send a variable indicating how many  “See Also”  will follow
      $vlNbSee:=Records in selection([CM See Also])
      SEND VARIABLE($vlNbSee)
 //  Send the [See Also] records (if any)
      For($vlSee;1;$vlNbSee)
         SEND RECORD([CM See Also])
         NEXT RECORD([CM See Also])
      End for
 // Go to the next [Commands] record and continue the export
      NEXT RECORD([Commands])
   End for
   SET CHANNEL(11) // Close the document
End if
```

Here is the (simplified) project method for importing the documentation:

```
 // CM_IMPORT_SEL project method

SET CHANNEL(10;"") // Let's user open an existing document
If(OK=1) // If a document was open
   RECEIVE VARIABLE($vsTag) // Try receiving the expected tag variable
   If($vsTag="4DV6COMMAND@") // Did we get the right tag?
      $CurLang:=Substring($vsTag;Length($vsTag)-1) // Extract language from the tag
      If(($CurLang="US")|($CurLang="FR")) // Did we get a valid language
         RECEIVE VARIABLE($vlNbCmd) // How many commands are there in this document?
         If($vlNbCmd>0) // If at least one
            For($vlCmd;1;$vlNbCmd) // For each archived [Commands] record
 // Receive the record
               RECEIVE RECORD([Commands])
 // Call a subroutine that saves the new record or copies its values
```

```
    // into an already existing record
                CM_IMP_CMD($CurLang)
    // Receive the number of parameters (if any)
                RECEIVE VARIABLE($vlNbParm)
                If($vlNbParm>=0)
    // Call a subroutine that calls RECEIVE RECORD then saves the new records
    // or copies them into already existing records
                   CM_IMP_PARM($vlNbParm;$CurLang)
                End if
    // Receive the number of  “See Also”  (if any)
                RECEIVE VARIABLE($vlNbSee)
                If($vlNbSee>0)
    // Call a subroutine that calls RECEIVE RECORD then saves the new records
    // or copies them into already existing records
                   CM_IMP_SEEA($vlNbSee;$CurLang)
                End if
             End for
          Else
             ALERT("The number of commands in this export document is invalid.")
          End if
       Else
          ALERT("The language of this export document is unkown.")
       End if
    Else
       ALERT("This document is NOT a Commands export document.")
    End if
    SET CHANNEL(11) // Close document
End if
```

Note that we do not test the OK variable while receiving the data nor try to catch the errors. However, because we stored variables in the document that describes the document itself, if these variables, once received, made sense, the probability for an error is very low. If for instance a user opens a wrong document, the first test stops the operation right away.

## System variables and sets

The OK system variable is set to 1 if the record is received. Otherwise, the OK system variable is set to 0.

# RECEIVE VARIABLE

RECEIVE VARIABLE ( variable )

| Parameter | Type | | Description |
|---|---|---|---|
| variable | Variable | ⇨ | Variable in which to receive |

## Description

**RECEIVE VARIABLE** receives *variable*, which was previously sent by **SEND VARIABLE** from the document or serial port previously opened by **SET CHANNEL**.

In interpreted mode, if the variable does not exist prior to the call to **RECEIVE VARIABLE**, the variable is created, typed and assigned according to what has been received. In compiled mode, the variable must be of the same type as what is received. In both cases, the contents of the variable are replaced with what is received.

**Notes:**

1. If you receive a variable from a document using this command, the document must have been opened using the **SET CHANNEL** command. You cannot use **RECEIVE VARIABLE** with a document opened with **Open document**, **Create document** or **Append document**.
2. This command does not support array variables. If you want to send and receive arrays from a document or over a serial port, use the **BLOB Commands**.
3. During the execution of **RECEIVE VARIABLE**, the user can interrupt the reception by pressing **Ctrl-Alt-Shift** (Windows) or **Command-Option-Shift** (Macintosh). This interruption generates an error -9994 that you can catch with an error-handling method installed using **ON ERR CALL**. Usually, you only need to handle the interruption of a reception while communicating over a serial port.

## Example

See example for the **RECEIVE RECORD** command.

## System variables and sets

The OK system variable is set to 1 if the variable is received. Otherwise, the OK system variable is set to 0.

# SEND PACKET

SEND PACKET ( {docRef ;} packet )

| Parameter | Type | | Description |
|---|---|---|---|
| docRef | DocRef | ⇒ | Document reference number, or Current channel (serial port or document) |
| packet | String, BLOB | ⇒ | String or BLOB to be sent |

## Description

**SEND PACKET** sends a packet to a serial port or to a document. If *docRef* is specified, the packet is written to the document referenced by *docRef*. If *docRef* is not specified, the packet is written to the serial port or document previously opened by the **SET CHANNEL** command.

A *packet* is just a piece of data, generally a string of characters.
You can also pass a BLOB in *packet*. This allows you to bypass the constraints related to encoding for characters sent in text mode (see example 2).

**Note:** When you pass a BLOB in *packet*, the command does not take into account any character set defined by the **USE CHARACTER SET** command. The BLOB is sent without any modification.

Before you use **SEND PACKET**, you must open a serial port or a document with **SET CHANNEL**, or open a document with one of the document commands.

When writing to a document, the first **SEND PACKET** begins writing at the beginning of the document unless the document was opened with **USE CHARACTER SET**. Until the document is closed, each subsequent packet is appended to any previously sent packets.

**Note:** This command is useful for a document opened with **SET CHANNEL**. On the other hand, for a document opened with **Open document**, **Create document** or **Append document**, you can use the commands **Get document position** and **SET DOCUMENT POSITION** to get and change the location in the document where the next writing (**SEND PACKET**) or reading (**RECEIVE PACKET**) will occur.

## Example 1

The following example writes data from fields to a document. It writes the fields as fixed-length fields. Fixed-length fields are always of a specific length. If a field is shorter than the specified length, the field is padded with spaces. (That is, spaces are added to make up the specified length.) Although the use of fixed-length fields is an inefficient method of storing data, some computer systems and applications still use them:

```
$vhDocRef :=Create document("") ` Create a document
If(OK=1) ` Was the document created?
   For($vlRecord;1;Records in selection([People])) ` Loop once for each record
   ` Send a packet. Create the packet from a string of 15 spaces containing the first name field
      SEND PACKET($vhDocRef;Change string(15*Char(SPACE);[People]First;1))
   ` Send a second packet. Create the packet from a string of 15 spaces containing the last name field
   ` This could be in the first SEND PACKET, but is separated for clarity
      SEND PACKET($vhDocRef;Change string(15*Char(SPACE);[People]Last;1))
      NEXT RECORD([People])
   End for
   ` Send a Char(26), which is used as an end-of-file marker for some computers
   SEND PACKET($vhDocRef;Char(SUB ASCII code))
   CLOSE DOCUMENT($vhDocRef) ` Close the document
End if
```

## Example 2

This example illustrates the sending and retrieval of extended characters via a BLOB in a document:

```
C_BLOB($send_blob)
C_BLOB($receive_blob)
TEXT TO BLOB("ázérty";$send_blob;UTF8 text without length)
SET BLOB SIZE($send_blob;16;255)
```

```
$send_blob{6}:=0
$send_blob{7}:=1
$send_blob{8}:=2
$send_blob{9}:=3
$send_blob{10}:=0
$vlDocRef:=Create document("blob.test")
If(OK=1)
   SEND PACKET($vlDocRef;$send_blob)
   CLOSE DOCUMENT($vlDocRef)
End if
$vlDocRef:=Open document(document)
If(OK=1)
   RECEIVE PACKET($vlDocRef;$receive_blob;65536)
   CLOSE DOCUMENT($vlDocRef)
End if
```

# SEND RECORD

SEND RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to send the current record, or Default table, if omitted |

## Description

**SEND RECORD** sends the current record of *aTable* to the serial port or document opened by the **SET CHANNEL** command. The record is sent with a special internal format that can be read only by **RECEIVE RECORD**. If no current record exists, **SEND RECORD** has no effect.

The complete record is sent. This means that pictures and BLOBs stored in or with the record are also sent.

**Important:** When records are being sent and received using **SEND RECORD** and **RECEIVE RECORD**, the source table structure and the destination table structure must be compatible. If they are not, 4D will convert values according to the table definitions when **RECEIVE RECORD** is executed.

**Note:** If you send a record to a document using this command, the document must have been opened using the **SET CHANNEL** command. You cannot use **SEND RECORD** with a document opened with **Open document**, **Create document** or **Append document**.

**Compatibility note:** Beginning with version 11 of 4D, this command no longer supports subtables.

## Example

See example for the **RECEIVE RECORD** command.

# SEND VARIABLE

SEND VARIABLE ( variable )

| Parameter | Type | | Description |
|---|---|---|---|
| variable | Variable | ⇨ | Variable to send |

## Description

**SEND VARIABLE** sends *variable* to the document or serial port previously opened by **SET CHANNEL**. The variable is sent with a special internal format that can be read only by **RECEIVE VARIABLE**. **SEND VARIABLE** sends the complete variable (including its type and value).

**Notes:**

1. If you send a variable to a document using this command, the document must have been opened using the **SET CHANNEL** command. You cannot use **SEND VARIABLE** with a document opened with **Open document**, **Create document** or **Append document**.
2. This command does not support array variables. If you want to send and receive arrays from a document or over a serial port, use the **BLOB Commands**.

## Example

See example for the **RECEIVE RECORD** command.

# SET CHANNEL

SET CHANNEL ( port ; settings )

| Parameter | Type | | Description |
|---|---|---|---|
| port | Longint | ⇒ | Serial port number |
| settings | Longint | ⇒ | Serial port settings |

SET CHANNEL ( operation ; document )

| Parameter | Type | | Description |
|---|---|---|---|
| operation | Longint | ⇒ | Document operation to perform |
| document | String | ⇒ | Document name |

## Description

The **SET CHANNEL** command opens a serial port or a document. You can open only one serial port or one document at a time with this command. To close an opened serial port, pass **SET CHANNEL** (11).

**Historical Note:** This command was originally the first 4D command used for working with serial ports and documents on disks. Since that time, new commands have been added. Today, you will typically work with documents on disk using the commands **Open document**, **Create document** and **Append document**. With these commands, you can read and write characters to and from documents using **Create document** or **RECEIVE PACKET** (these commands work with **SET CHANNEL**, too). However, if you want to use the commands **SEND VARIABLE**, **RECEIVE VARIABLE**, **SEND RECORD** and **RECEIVE RECORD**, you must use **SET CHANNEL** to access the document on disk.

The description of **SET CHANNEL** is composed of two sections:

- Working with Serial Ports
- Working with Documents

## Working with Serial Ports: SET CHANNEL (port;settings)

The first form of the **SET CHANNEL** command opens a serial port, setting the protocol and other port information. Data can be sent with **SEND PACKET**, **SEND RECORD** or **SEND VARIABLE**, and received with **RECEIVE BUFFER**, **RECEIVE PACKET**, **RECEIVE VARIABLE** or **RECEIVE RECORD**.
The first parameter, *port*, selects the port and the protocol. You can address up to 99 serial ports (one at a time).
The following table lists the values for *port*:

| Value for port | Description |
|---|---|
| 0 | Printer port (Macintosh) or COM2 (PC) with no protocol |
| 1 | Modem port (Macintosh) or COM1 (PC) with no protocol |
| 20 | Printer port (Macintosh) or COM2 (PC) with sofware protocol such as XON/XOFF |
| 21 | Modem port (Macintosh) or COM1 (PC) with sofware protocol such as XON/XOFF |
| 30 | Printer port (Macintosh) or COM2 (PC) with hardware protocol such as RTS/CTS |
| 31 | Modem port (Macintosh) or COM1 (PC) with hardware protocol such as RTS/CTS |
| 101 to 199 | Serial communication with no protocol |
| 201 to 299 | Serial communication with software protocol such as XON/XOFF |
| 301 to 399 | Serial communication with hardware protocol such as RTS/CTS |

**Important:** The value you pass in *port* must refer to an existing serial COM port recognized by the operating system. For example, in order to be able to use the values 101, 103 and 125, the serial ports COM1, COM3 and COM25 must have been set up correctly.

### Note on serial ports

In a standard configuration Mac OS and Windows support two serial ports: on Mac OS, the modem port and the printer port; on Windows, the COM1 and COM2 ports. However, additional serial ports can be added by the use of extension boards. Originally, 4D only adressed two standard serial ports and it was only later that the support of additional ports was implemented. For compatibility reasons, both addressing systems were kept.

- If you want to address a standard serial port (printer/COM2 or modem/COM1), you can either pass in the *port* parameter one of the following values 0, 1, 20, 21, 30 and 31 (that corresponds to the old addressing method), or a value greater than 100 (please see the following explanation).
- If you want to address additional serial ports, you need to pass the value N+100 (where N is the value of the port to address). You may also consider adding 100 or 200 to the value mentioned above (N+100), if you want to select respectively a software or a hardware protocol.

### Example 1

If you want to use the printer/COM2 port with no protocol, you can use one of the following syntaxes:

```
SET CHANNEL(0;param)
```

or

```
SET CHANNEL(102;param)
```

### Example 2

If you want to use the modem/COM1 port with the XON/XOFF protocol, you can use one of the following syntaxes:

```
SET CHANNEL(21;param)
```

or

```
SET CHANNEL(201;param)
```

### Example 3

If you want to use the COM 25 port with the RTS/CTS protocol, you need to use the following syntax:

```
SET CHANNEL(325;param)
```

The *settings* parameter sets the speed, number of data bits, number of stop bits, and parity. You determine the value for *settings* by adding the speed, data bits, stop bits, and parity values as listed in the following table. For example, to set 1200 baud, 8 data bits, 1 stop bit, and no parity, you would add 94 + 3072 + 16384 + 0 = 19550. You would then use 19550 as the value of the *setup* parameter.

| | Value to accumulate in settings parameter | Description |
|---|---|---|
| Speed | 380 | 300 |
| (in baud) | 189 | 600 |
| | 94 | 1200 |
| | 62 | 1800 |
| | 46 | 2400 |
| | 30 | 3600 |
| | 22 | 4800 |
| | 14 | 7200 |
| | 10 | 9600 |
| | 4 | 19200 |
| | 2 | 28800 |
| | 1 | 38400 |
| | 0 | 57600 |
| | 1022 | 115200 |
| | 1021 | 230400 |
| Data bits | 0 | 5 |
| | 2048 | 6 |
| | 1024 | 7 |
| | 3072 | 8 |
| Stop bits | 16384 | 1 |
| | –32768 | 1.5 |
| | –16384 | 2 |
| Parity | 0 | None |
| | 4096 | Odd |
| | 12288 | Even |

**Tip:** The various numeric values to be accumulated and passed in *port* and *settings* (but not including the values for COM1...COM99) are available as predefined constants in the theme **Communications** within the Design environment Explorer windows. For COM1...COM99, use numeric literals.

## Working with Documents on Disk: SET CHANNEL(operation;document)

The second form of the **SET CHANNEL** command allows you to create, open, and close a document. Unlike the **System Documents** commands, it can open only one document at a time. The document can be read from or written to.

The *operation* parameter specifies the operation to be performed on the document specified by *document*. The following table lists the values of *operation* and the resulting operation with different values for *document*. The first column lists the allowed values for *operation*. The second column lists the allowed values for *document*. The third column lists the resulting operation.

For example, to display an Open File dialog box to open a text file, you would use the following line:

```
SET CHANNEL(13;"")
```

| Operation | Document | Result |
|---|---|---|
| 10 | String | Opens the document specified by String.<br>If the document doesn't exist, the document is opened and created. |
| 10 | "" (empty string) | Displays the Open File dialog box to open a file.<br>All file types are displayed. |
| 11 | none | Closes an open file. |
| 12 | "" (empty string) | Displays the Save File dialog box to create a new file. |
| 13 | "" (empty string) | Displays the Open File dialog box to open a file. Only text file types are displayed. |

All of the operations in this table set the *Document* system variable if appropriate. They also set the OK system variable to 1 if the operation was successful. Otherwise, the OK system variable is set to 0.

### Example 4

See examples for the **RECEIVE BUFFER**, **SET TIMEOUT** and **RECEIVE RECORD** commands.

# SET TIMEOUT

SET TIMEOUT ( seconds )

| Parameter | Type | | Description |
|---|---|---|---|
| seconds | Longint | ⇨ | Seconds until the timeout |

## Description

**SET TIMEOUT** specifies how much time a serial port command has to complete. If the serial port command does not complete within the specified time, *seconds*, the serial port command is canceled, an error -9990 is generated, and the OK system variable is set to 0. You can catch the error with an error-handling method installed using **ON ERR CALL**.

Note that the time is the total time allowed for the command to execute, not the time between characters received. To cancel a previous setting and stop monitoring serial port communication, use a setting of 0 for *seconds*.

The commands that are affected by the timeout setting are:

- **RECEIVE PACKET**
- **RECEIVE RECORD**
- **RECEIVE VARIABLE**

## Example

The following example sets the serial port to receive data. It then sets a time-out. The data is read with **RECEIVE PACKET**. If the data is not received in time, an error occurs:

```
SET CHANNEL(MacOS serial port;Speed 9600+Data bits 8+Stop bits one+Parity none) ` Open Serial Port
SET TIMEOUT(10) ` Set the timeout for 10 seconds
ON ERR CALL("CATCH COM ERRORS") ` Do not let the method being interrupted
RECEIVE PACKET(vtBuffer;Char(13)) ` Read until a carriage return is met
If(OK=0)
   ALERT("Error receiving data.")
Else
   [People]Name:=vtBuffer ` Save received data in a field
End if
ON ERR CALL("")
```

# USE CHARACTER SET

USE CHARACTER SET ( map {; mapInOut} )

| Parameter | Type | | Description |
|---|---|---|---|
| map | String, Operator | ⇒ | Name of character set to use, or * to reset to default character set |
| mapInOut | Longint | ⇒ | 0 = Output map 1 = Input map, If omitted, output map |

## Description

**USE CHARACTER SET** modifies the character set used by 4D during data transfer between the database and a document or a serial port for the current process. Transfer operations include the import and export of text, DIF, and SYLK files. A character map also works on data transferred with **SEND PACKET**, **RECEIVE PACKET** (for text type packets) and **RECEIVE BUFFER**. It has no effect on transfers of data done with **SEND RECORD**, **SEND VARIABLE**, **RECEIVE RECORD**, **SEND PACKET**, **RECEIVE PACKET** (for BLOB type packets) and **RECEIVE VARIABLE**.

The *map* parameter must correspond to the “IANA” name of the character set to be used, or to one of its aliases. For example, the names “iso-8859-1” or “utf-8” are both valid names, as well as the aliases “latin1” or “l1”. For more information about these names, please refer to the following address: *http://www.iana.org/assignments/character-sets*. Examples if IANA names are also provided in the description of the **CONVERT FROM TEXT** command.

If *mapInOut* is 0, the map is set for exporting. If *mapInOut* is 1, the map is set for importing. If you do not pass the *mapInOut* parameter, the export map is used by default.

When the * parameter is passed, the default character set is restored (import or export map depending on the value of *mapInOut*).
In 4D, the default character set is UTF-8.

## Example

The following example (Unicode mode) uses the UTF-16 character set to export a text, then the default character set is restored:

```
USE CHARACTER SET("UTF-16LE";0) ` Use the UTF-16 'Little Endian' character set
EXPORT TEXT([MyTable];"MyText") ` Export data through the map
USE CHARACTER SET(*;0) ` Restore the default character set
```

## System variables and sets

The OK system variable is set to 1 if the map is loaded correctly. Otherwise, it is set to 0.

# Compiler

- Compiler Commands
- Using Compiler Directives
- Typing Guide
- Syntax Details
- Optimization Hints
- Error messages
- C_BLOB
- C_BOOLEAN
- C_DATE
- C_LONGINT
- C_OBJECT
- C_PICTURE
- C_POINTER
- C_REAL
- C_TEXT
- C_TIME
- IDLE
- *_o_C_GRAPH*
- *_o_C_INTEGER*
- *_o_C_STRING*

# Compiler Commands

The integrated compiler of 4D translates your database applications into assembly level instructions. The advantages of the compiler are:

- **Speed:** Your database can run from 3 to 1,000 times faster.
- **Code checking:** Your database application is scanned for the consistency of code. Both logical and syntactical conflicts are detected.
- **Protection:** once your database is compiled, you can delete the interpreted code, Then, the compiled database is functionally identical to the original, except that the structure and procedures cannot be viewed or modified, deliberately or inadvertently.
- **Stand-alone double-clickable applications:** compiled databases can also be transformed into stand-alone applications (.EXE files) with their own icon.
- **Execution in preemptive mode**: only compiled code can be executed in a preemptive process.

For a description of the operation of the 4D compiler, refer to the Design Reference manual.

The commands in this theme relate to the use of the compiler. They enable you to normalize data types throughout your database. The **IDLE** command is specifically used in compiled databases.

| | | |
|---|---|---|
| **C_BLOB** | **C_REAL** | **C_TEXT** |
| **C_BOOLEAN** | **C_LONGINT** | **C_DATE** |
| **C_POINTER** | **C_PICTURE** | **C_TIME** |
| **C_OBJECT** | **IDLE** | |

**Compatibility note:** The obsolete **_o_C_GRAPH**, **_o_C_INTEGER** and **_o_C_STRING** commands must no longer be used.

These commands, except **IDLE**, declare variables and cast them as a specified data type. Declaring variables resolves ambiguities concerning a variable's data type. If a variable is not declared with one of these commands, the compiler attempts to determine a variable's data type. The data type of a variable used in a form is often difficult for the compiler to determine. Therefore, it is especially important that you use these commands to declare a variable used in a form.

**Note**: To save time, you can use the option for generating and updating typing methods (called "Compiler methods") found in the compiler window. This option automatically creates typing methods that take stock of and assign a type to all of the variables used in the database.

**Arrays** are variables that must follow the same rules as standard variables with respect to compilation. The array declaration commands are grouped together in the "**Arrays**" theme.

## General rules about writing code that will be compiled

- You must not give the same name to more than one method or variable. You cannot have a method with the same name as a variable.
- Variable indirection as used in 4D version 1 is not allowed. You cannot use alpha indirection, with the section symbol (§), to indirectly reference variables. Nor can you use numeric indirection, with the curly braces ({...}), for this purpose. Curly braces can only be used when accessing specific elements of an array that has been declared. However, you can use parameter indirection.
- You can't change the data type of any variable or array.
- You can't change a one-dimensional array to a two-dimensional array, or change a two-dimensional array to a one-dimensional array.
- You can't change the length of string variables or of elements in string arrays.
- Although the compiler will type the variable for you, you should specify the data type of a variable by using compiler directives where the data type is ambiguous, such as in a form.
- Another reason to explicitly type your variables is to optimize your code. This rule applies especially to any variable used as a counter. Use variables of a long integer data type for maximum performance.

- To clear a variable (initialize it to null), use **CLEAR VARIABLE** with the name of the variable. Do not use a string to represent the name of the variable in the **CLEAR VARIABLE** command.
- The **Undefined** function will always return *False*. Variables are always defined.
- Numeric operations on long integer and integer variables are usually much faster than operations on the default numeric type (real).
- If you have checked the "Can be run in preemptive processes" property for the method, the code must not call any thread-unsafe commands or other thread-unsafe methods.

These principles are detailed in the following sections:

- **Using Compiler Directives**, explains when and where to write compiler directives,
- **Typing Guide**, describes the different types of conflicts that may occur during the compilation of 4D databases,
- **Syntax Details**, provides additional information concerning several 4D commands,
- **Optimization Hints**, offers hints to accelerate the running of applications in compiled mode,
- **Writing a thread-safe method**.

## Example 1

The following are some basic variable declarations for the compiler:

```
C_BLOB(vxMyBlob) // The process variable vxMyBlob is declared as a variable of type BLOB
C_BOOLEAN(<>OnWindows) // The interprocess variable <>OnWindows is declared as a variable of type Boolean
C_DATE($vdCurDate) // The local variable $vdCurDate is declared as a variable of type Date
C_LONGINT(vg1;vg2;vg3) // The 3 process variables vg1, vg2 and vg3 are declared as variables of type longint
```

## Example 2

In the following example, the ***OneMethodAmongOthers*** project method declares 3 parameters:

```
// OneMethodAmongOthers Project Method
// OneMethodAmongOthers ( Real ; Date { ; Long } )
// OneMethodAmongOthers ( Amount ; Date { ; Ratio } )

C_REAL($1) // 1st parameter is of type Real
C_DATE($2) // 2nd parameter is of type Date
C_LONGINT($3) // 3rd parameter is of type Long Integer

// ...
```

## Example 3

In the following example, the *Capitalize* project method accepts a text parameter and returns a text result:

```
// Capitalize Project Method
// Capitalize ( Text ) -> Text
// Capitalize ( Source string ) -> Capitalized string

C_TEXT($0;$1)
$0:=Uppercase(Substring($1;1;1))+Lowercase(Substring($1;2))
```

## Example 4

In the following example, the project method ***SEND PACKETS*** accepts a time parameter followed by a variable number of text parameters:

```
` SEND PACKETS Project Method
` SEND PACKETS ( Time ; Text { ; Text2... ; TextN } )
` SEND PACKETS ( docRef ; Data { ; Data2... ; DataN } )
```

```
C_TIME($1)
C_TEXT(${2})
C_LONGINT($vlPacket)

For($vlPacket;2;Count parameters)
   SEND PACKET($1;${$vlPacket})
End for
```

## Example 5

In the following example, the ***COMPILER_Param_Predeclare28*** project method predeclares the syntax of other project methods for the compiler:

```
 // COMPILER_Param_Predeclare28 Project Method

C_REAL(OneMethodAmongOthers;$1) // OneMethodAmongOthers ( Real ; Integer { ; Long } )
C_DATE(OneMethodAmongOthers;$2) // ...
C_LONGINT(OneMethodAmongOthers;$3) // ...
C_TEXT(Capitalize;$0;$1) // Capitalize ( Text) -> Text
C_TIME(SEND PACKETS;$1) // SEND PACKETS ( Time ; Text { ; Text2... ; TextN } )
C_TEXT(SEND PACKETS;${2}) // ...
```

## Using Compiler Directives

### Data types of variables

4D has three categories of variables:

- Local variables,
- Process variables,
- Interprocess variables.

For more information about this point, refer to the **Variables** section. Process and interprocess variables are structurally the same for the compiler.

Since the compiler cannot determine the process in which the variable will be used, process variables should be used with more care than interprocess variables. All process variables are systematically duplicated when a new process begins. A process variable can have a different value in each process, but it has the same type for the entire database.

#### Types of Variables

All variables have a type. As described in the **Data Types** section, there are several different types of variables:

| |
|---|
| Boolean |
| Date |
| Longint |
| Graph |
| Time |
| Picture |
| Number (or Real) |
| Pointer |
| Text |
| BLOB |
| Object |

Here are the different types of arrays:

| |
|---|
| Boolean Array |
| Date Array |
| Integer Array |
| Longint Array |
| Picture Array |
| Real Array |
| Time Array |
| Object Array |
| Pointer Array |
| BLOB Array |
| Text Array |

**Notes:**
- The Object type is available since 4D v14.
- The former Alpha (fixed string) and Integer types are no longer used for variables. In existing code, they are automatically redirected to Text and Longint types.

#### Creation of the symbol table

In interpreted mode, a variable can have more than one data type. This is possible because the code is interpreted rather than compiled. 4D interprets each statement separately and comprehends its context. When you work in a compiled environment, the situation is different. While interpretation is performed line by line, the compilation process looks at a database in its entirety.

The compiler's approach is the following:

- The compiler systematically analyzes the objects in the database. The objects are database, project, form, trigger and object methods.
- The compiler scans the objects to determine the data type of each variable used in the database, and it generates the table of variables and methods (the symbol table).
- Once it has established the data types of all variables, the compiler translates (compiles) the database. However, it cannot compile the database unless it can determine the data type for each of the variables.

If the compiler comes across the same variable name and two different data types, it has no reason to favor any particular one. In other words, in order to type an object and give it a memory address, the compiler must know the precise identity of that object (i.e., its name and its data type). The compiler determines its size from the data type. For every compiled database, the compiler creates a map that lists, for each variable, its name (or identifier), its location (or memory address), and the space it occupies (indicated by its data type). This map is called the symbol table. An option in the Preferences lets you choose whether to generate this table in the form of a file during compilation.
This map is also used for the automatic generation of compiler methods.

### Variables typed by the compiler

If you want the compiler to check the typing of your variables or to type them itself, it is easy to place a compiler directive for this purpose. You can choose between two different possibilities, depending on your working methods:

- Either use the directive in the method in which the variable first appears, depending on whether it is a local, proces or interprocess variable. Be sure to use the directive the very first time you use the variable, in the first method to be executed. Keep in mind that during compilation, the compiler takes the methods in the order of their creation in 4D, and not in the order in which they are displayed in the Explorer.
- Or, if you are systematic in your approach, group all the process and interprocess variables with the different compiler directives in the **On Startup database method** or in a method called by the **On Startup database method**.
  For local variables, group the directives at the beginning of the method in which they appear.

### Default values

When variables are typed by means of a compiler directive, they receive a default value, which they will keep during the session as long as they have not been assigned.

The default value depends on the variable type and category, its execution context (interpreted or compiled), as well as, for compiled mode, the compilation options defined on the **Compiler page** of the Database settings:

- Process and interprocess variables are always set "to zero" (which means, depending on the case, "0", an empty string, an empty Blob, a Nil pointer, a blank date (00-00-00), etc.)
- Local variables are set:
  - in interpreted mode: to zero
  - in compiled mode, depending on the **Initialize local variables** option of the Database settings:
    - to zero when "to zero" is chosen,
    - to a set random value when "to a random value" is chosen (0x72677267 for numbers and times, always True for Booleans, the same as "to zero" for the others),
    - to a random value (for numbers) when "no" is chosen.

The following table illustrates these default values:

| Type | Interprocess | Process | Local interpreted | Local compiled "to zero" | Local compiled "random" | Local compiled "no" |
|---|---|---|---|---|---|---|
| Booleen | False | False | False | False | True | True (varies) |
| Date | 00-00-00 | 00-00-00 | 00-00-00 | 00-00-00 | 00-00-00 | 00-00-00 |
| Longint | 0 | 0 | 0 | 0 | 1919382119 | 909540880 (varies) |
| Graph | 0 | 0 | 0 | 0 | 1919382119 | 775946656 (varies) |
| Time | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 533161:41:59 | 249345:34:24 (varies) |
| Picture | picture size=0 | picture size=0 | picture size=0 | picture size=0 | picture size=0 | picture size=0 |
| Real | 0 | 0 | 0 | 0 | 1.250753659382e+243 | 1.972748538022e-217 (varies) |
| Pointer | Nil=true | Nil=true | Nil=true | Nil=true | Nil=true | Nil=true |
| Text | "" | "" | "" | "" | "" | "" |
| Blob | Blob size=0 | Blob size=0 | Blob size=0 | Blob size=0 | Blob size=0 | Blob size=0 |
| Object | undefined | undefined | undefined | undefined | undefined | undefined |

## When to use compiler directives

Compiler directives are useful in two cases:

- The compiler is unable to determine the data type of a variable from its context,
- You do not want the compiler to determine a variable's type from its use.

Furthermore, using compiler directives allows you to reduce compilation time.

### Cases of ambiguity

Sometimes the compiler cannot determine the data type of a variable. Whenever it cannot make a determination, the compiler generates an appropriate error message.
There are three major causes that prevent the compiler from determining the data type: multiple data types, ambiguity on a forced deduction and the inability to determine a data type.

#### Multiple data types

If a variable has been retyped in different statements in the database, the compiler generates an error that is easy to fix. The compiler selects the first variable it encounters and arbitrarily assigns its data type to the next occurrence of the variable having the same name but a different data type.

Here is a simple example:

in method A,

```
Variable:=True
```

in method B,

```
Variable:="The moon is green"
```

If method A is compiled before method B, the compiler considers the statement *Variable:="The moon is green"* as a data type change in a previously encountered variable. The compiler notifies you that retyping has occurred. It generates an error for you to correct. In most cases, the problem can be fixed by renaming the second occurrence of the variable.

#### Ambiguity on a forced deduction

Sometimes, due to a sequence, the compiler can deduce that an object's type is not the proper type for it. In this case, you must explicitly type the variable with a compiler directive.

Here is an example using the default values for an active object:

In a form, you can assign default values for the following objects: combo boxes, pop-up menus, tab controls, drop-down lists, menu/drop-down lists and scrollable areas using the Edit button for the **Value List** (under the Entry Control theme of the Property List) (for more information, refer to the 4D Design Reference manual). The default values are automatically loaded into an array whose name is the same as the name of the object.

If the object is not used in a method, the compiler can deduce the type, without ambiguity, as a text array.

However, if a display initialization must be performed, the sequence could be:

```
Case of
   :(Form event=On Load)
      MyPopUp:=2
      ...
End case
```

In this case, the ambiguity appears--when parsing methods, the compiler will deduce a Real data type for the object MyPopUp. In this case, it is necessary to explicitly declare the array in the form method or in a compiler method:

```
Case of
   :(Form event=On Load)
      ARRAY TEXT(MyPopUp;2)
      MyPopUp:=2
      ...
End case
```

**Inability to determine a data type**

This case can arise when a variable is used without having been declared, within a context that does not provide information about its data type. Here, only a compiler directive can guide the compiler.
This phenomenon occurs primarily within four contexts:

- when pointers are used,
- when you use a command with more than one syntax,
- when you use a command having optional parameters of different data types,
- when you use a 4D method called via a URL.

*Pointers*

A pointer cannot be expected to return a data type other than its own.
Consider the following sequence:

```
Var1:=5.2(1)
Pointer:=->Var1(2)
Var2:=Pointer->(3)
```

Although (2) defines the type of variable pointed to by Pointer, the type of Var2 is not determined. During compilation, the compiler can recognize a pointer, but it has no way of knowing what type of variable it is pointing to. Therefore it cannot deduce the data type of Var2. A compiler directive is needed, for example *C_REAL(Var2)*.

*Multi-syntax commands*

When you use a variable associated with the function **Year of**, the variable can only be of the data type Date, considering the nature of this function. However, things are not always so simple. Here is an example:
The **GET FIELD PROPERTIES** command accepts two syntaxes:
**GET FIELD PROPERTIES**(tableNo;fieldNo;type;length;index)
**GET FIELD PROPERTIES**(fieldPointer;type;length;index)

When you use a multi-syntax command, the compiler cannot guess which syntax and parameters you have selected. You must use compiler directives to type variables passed to the command, if they are not typed according to their use elsewhere in the database.

*Commands with optional parameters of different data types*

When you use a command that contains several optional parameters of different data types, the compiler cannot determine which optional parameters have been used. Here is an example:
The **GET LIST ITEM** command accepts two optional parameters; the first as a Longint and the other as a Boolean.
The command can thus either be used as follows:
**GET LIST ITEM**(list;itemPos;itemRef;itemText;sublist;expanded)
or like this:
**GET LIST ITEM**(list;itemPos;itemRef;itemText;expanded)
You must use compiler directives to type optional variables passed to the command, if they are not typed according to their use elsewhere in the database.

*Methods called via URLs*

If you write 4D methods that need to be called via a URL, and if you do not use $1 in the method, you must explicitly

declare the text variable $1 with the following sequence:
**C_TEXT**($1)
In fact, the compiler cannot determine that the 4D method will be called via a URL.

### Reducing time needed to compile

If all the variables used in the database are explicitly declared, it is not necessary for the compiler to check the typing. In this case, you can set the options so that the compiler only executes the translation phase of the method. This saves at least 50% in compilation time.

### Optimizing code

You can speed up your methods by using compiler directives. For more details on this subject, refer to the **Optimization Hints** section. To give a simple example, suppose you need to increment a counter using a local variable. If you do not declare the variable, the compiler assumes that is a Real. If you declare it as a Longint, the compiled database will perform more efficiently. On a PC, for instance, a Real takes 8 bytes, but if you type the counter as a Longint, it only uses 4 bytes. Incrementing an 8-byte counter obviously takes longer than incrementing a 4-byte one.

## Where to place your compiler directives

Compiler directives can be handled in two different ways, depending on whether or not you want the compiler to type your variables.

### Typing variables

The compiler must respect the identification criteria of the variables.

There are two possibilities:

1) If the variables are not typed, the compiler can do it for you automatically. Whenever possible––as long as there is no ambiguity––the compiler determines a variable's type from the way it is used. For example, if you write:

```
V1:=True
```

the compiler determines that variable V1 is of data type Boolean.

By the same token, if you write:

```
V2:="This is a sample phrase"
```

the compiler determines that V2 is a Text type variable.

The compiler is also capable of establishing the data type of a variable in less straightforward situations:

```
V3:=V1 `V3 is of the same type as V1
V4:=2*V2 `V4 is of the same type as V2
```

The compiler also determines the data type of your variables according to calls to 4D commands and according to your methods. For example if you pass a Boolean type parameter and a Date type parameter to a method, the compiler assigns the Boolean type and the Date type to the local variables $1 and $2 in the called method.

When the compiler determines the data type by inference, unless indicated otherwise in the Preferences, it never assigns the limiting data types: Integer, Longint or String. The default type assigned by the compiler is always the widest possible. For example, if you write:

```
Number:=4
```

the compiler assigns the Real data type to Number, even though 4 happens to be an integer. In other words, the compiler does not rule out the possibility that, under other circumstances, the variable's value might be 4.5.
If it is appropriate to type a variable as Integer, Longint or String, you can do so using a compiler directive. It is to your advantage to do so, because these data types occupy less memory and performing operations on them is faster.

If you have already typed your variables and are sure that your typing is coherent and complete, you may explicitly ask the compiler not to redo this work, using the compilation Preferences. In case your typing was not complete and exhaustive, at the time of compilation, the compiler will return errors requesting you to make the necessary modifications.

2) The compiler directive commands enable you to explicitly declare the variables used in your databases.

They are used in the following manner:

```
C_BOOLEAN(Var)
```

Through such directives, you inform the compiler to create a variable Var that will be a Boolean.
Whenever an application includes compiler directives, the compiler detects them and thus avoids guesswork.

A compiler directive has priority over deductions made from assignments or use.

Variables declared with the compiler directive **_o_C_INTEGER** are actually the same as those declared by the directive **C_LONGINT**. They are, in fact, long integers between -2147483648 and +2147483647.

### Variables typed by the developer

If you do not want the compiler to check your typing, you must give it a code to identify the compiler directives.
The convention to follow is:
Compiler directives for the process and interprocess variables and the parameters should be placed in one or more methods, the names of which begin with the key word **Compiler**.
By default, the compiler lets you automatically generate five types of Compiler methods, which group together the directives for variables, arrays and method parameters (for more information about this point, refer to the Design Reference manual).

**Note:** The syntax for declaring these parameters is the following:
*Directive (methodName;parameter)*. This syntax is not executable in interpreted mode.

**Particular parameters**

- Parameters received by database methods
  If these parameters have not been explicitly declared, they are typed by the compiler. Nevertheless, if you declare them, the declaration must be done inside the database methods.
  This parameter declaration cannot be written in a Compiler method.
  Example: **On Web Connection Database Method** receives six parameters, $1 to $6, of the data type Text. At the beginning of the database method, you must write: **C_TEXT**($1;$2;$3;$4;$5;$6)

- Triggers
  The $0 parameter (Longint), which is the result of a trigger, is typed by the compiler if the parameter has not been explicitly declared. Nevertheless, if you want to declare it, you must do so in the trigger itself.
  This parameter declaration cannot be written in a Compiler method.

- Objects that accept the “On Drag Over” form event
  The $0 parameter (Longint), which is the result of the “On Drag Over” form event, is typed by the compiler if the parameter has not been explicitly declared. Nevertheless, if you want to decalre it, you must do so in the object method.
  This parameter declaration cannot be written in a Compiler method.
  **Note:** The compiler does not initialize the $0 parameter. So, as soon as you use the On Drag Over form event, you must initialize $0. For example:

```
C_LONGINT($0)
If(Form event=On Drag Over)
   $0:=0
   ...
   If($DataType=Is picture)
      $0:=-1
   End if
   ...
End if
```

## A certain liberty permitted by the compiler

Compiler directives remove any ambiguity concerning data types. Although a certain rigor is necessary, this does not necessarily mean that the compiler is intolerant of any and every inconsistency.
For example, if you assign a real value to a variable declared as an Integer, the compiler does not regard either assignment as a type conflict and assigns the corresponding values according to your directives. So, if you write:

```
C_LONGINT(vInteger)
vInteger:=2.6
```

The compiler does not regard it as a data type conflict that will prevent compilation; instead, the compiler simply rounds off to the closest integer value (3 instead of 2.6).

# Typing Guide

This section describes the main causes of typing conflicts on variables, as well as ways to avoid them.

## Conflicts on simple variables

Simple data type conflicts can be summarized as follows:

- conflict between two uses,
- conflict between use and a compiler directive,
- conflict resulting from implicit retyping,
- conflict between two compiler directives.

### Conflicts between two uses

The simplest data type conflict is one that stems from a single variable name designating two different objects. Suppose that, in an application, you write:

```
Variable:=5
```

and that elsewhere, in the same application, you write:

```
Variable:=True
```

This generates a data type conflict. The problem can be solved by renaming one of the variables.

### Conflict between use and a compiler directive

Suppose that, in an application, you write:

```
Variable:=5
```

and that elsewhere, in the same application, you write:

```
C_BOOLEAN(Variable)
```

Since the compiler scans the directives first, it will type Variable as Boolean, but when it finds the statement:

```
Variable:=5
```

it detects a data type conflict. You can solve the problem by renaming your variable or modifying the compiler directive.

Using variables of different data types in one expression creates inconsistencies. The compiler points out incompatibilities. Here is a simple example:

```
vBool:=True `The compiler infers that vBoolean is data type Boolean
C_LONGINT(<>vInteger) `Declaration of an Longint by a compiler directive
<>vInteger:=3 `Command compatible with the compiler directive
Var:=<>vInteger+vBool `Operation containing variables with incompatible data types
```

### Conflict stemming from implicit retyping

Some functions return variables of a very precise data type. Assigning the result of one of such variables to a variable already typed differently will cause a data type conflict if you are not careful.

For example, in an interpreted application, you can write:

```
IdentNo:=Request("Identification Number") `IdentNo is data type Text
If(Ok=1)
   IdentNo:=Num(IdentNo) `IdentNo is data type Real
```

```
   QUERY([Contacts]Id=IdentNo)
End if
```

In this example, you create a type conflict in the third line. The solution consists in controlling the behavior of the variable. In some cases, you must create an intermediate variable that uses a different name. In other cases, such as this, your method can be structured differently:

```
IdentNo:=Num(Request("Identification Number")) `IdentNo is data type Real
If(Ok=1)
   QUERY([Contacts]Id=IdentNo)
End if
```

### Conflict between two compiler directives

Declaring the same variable through two conflicting compiler directives constitutes a retyping. If, in the same database, you write:

```
C_BOOLEAN(Variable)
C_TEXT(Variable)
```

the compiler detects the conflict and reports an error in the error file. Typically, you can solve the problem by renaming one of the variables.

### Note concerning local variables

Data type conflicts involving local variables are identical to those in process or interprocess variables. The only difference is that there must be consistency only within a specified method.

For process and interprocess variables, conflicts occur at the general leve of teh database. For local variables, conflicts occur at the level of the method. For example, you cannot write in the same method:

```
$Temp:="Flowers"
```

and then

```
$Temp:=5
```

However, you can write:

```
$Temp:="Flowers"
```

in method M1, and:

```
$Temp:=5
```

in method M2, because the scope of local variables is the method itself and not the entire database.

## Conflicts in arrays

Conflicts concerning an array are never size-related. As in uncompiled databases, arrays are managed dynamically. The size of an array can vary throughout methods, and you do not have to declare a maximum size for an array.

Therefore, you can size an array to null, add or remove elements, or delete the contents.

You should follow these guidelines when writing a database intended for compilation:

- Do not change data types of array elements,
- Do not change the number of dimensions of an array,
- For a String array, do not change character-string length.

### Changing data types of array elements

If you declare an array as an Integer array, it must remain an integer array throughout the database. It can never contain, for example, Boolean type elements.

If you write:

```
ARRAY INTEGER(MyArray;5)
ARRAY BOOLEAN(MyArray;5)
```

the compiler cannot type MyArray.

Just rename one of the arrays.

### Changing the number of dimensions of an array

In an uncompiled database, you can change the number of dimensions in an array. When the compiler sets up the symbol table, one-dimensional arrays and two-dimensional arrays are managed differently.

Consequently, you cannot redeclare a one-dimensional array as two-dimensional, or vice versa.

Therefore, in the same database, you cannot have:

```
ARRAY INTEGER(MyArray1;10)
ARRAY INTEGER(MyArray1;10;10)
```

However, you can write the following statements in the same application:

```
ARRAY INTEGER(MyArray1;10)
ARRAY INTEGER(MyArray2;10;10)
```

The number of dimensions in an array cannot be changed in a database. However, you can change the size of an array. You can resize one array of a two-dimensional array and write:

```
ARRAY BOOLEAN(MyArray;5)
ARRAY BOOLEAN(MyArray;10)
```

**Note:** A two-dimensional array is, in fact, a set of several one-dimensional arrays. For more information, refer to the **Two-dimensional Arrays** section.

### Implicit retyping

When using commands such as **COPY ARRAY**, **LIST TO ARRAY**, **ARRAY TO LIST**, **SELECTION TO ARRAY**, **SELECTION RANGE TO ARRAY**, **ARRAY TO SELECTION**, or **DISTINCT VALUES**, you may change, voluntarily or not, the data types of elements, or the number of dimensions. You will thus find yourself in one of the situations previously mentioned.

The compiler generates an error message; the required correction is usually quite obvious. Examples of implicit array retyping are provided in the **Syntax Details** section.

### Local arrays

If you want to compile a database that uses local arrays (arrays only visible by the methods that created them), you must explicitly declare them in 4D before using them.

Explicitly declaring an array means using a command of the type **ARRAY REAL**, **ARRAY INTEGER**, etc.

For example, if a method creates a local integer array of 10 elements, you should have the following line in your method:

```
ARRAY INTEGER($MyArray;10)
```

## Typing of variables created in forms

Variables created in a form (e.g., buttons, drop-down list boxes, and so forth) are always process or interprocess variables.

In an interpreted database, the data type of such variables is not important. However, in compiled applications, it may have to be taken into consideration. The rules are, nevertheless, quite clear:

- You can type form variables using compiler directives, or
- The compiler assigns it a default type that can be set in the compilation Preferences (see the Design Reference manual).

### Variables considered by default as Real

The following form variables are typed as Real by default:

| |
|---|
| Check box |
| 3D check box |
| Button |
| Highlight button |
| Invisible button |
| 3D button |
| Picture button |
| Button grid |
| Radio button |
| 3D radio button |
| Radio picture |
| Picture menu |
| Hierarchical pop-up menu |
| Hierarchical list |
| Ruler |
| Dial |
| Thermometer |
| List box (selection type) |

**Note:** The Ruler, Dial and Thermometer form variables are always typed as Reals, even if you choose Long integer as the Default Button Type in the Preferences.

For one of these variables, the only data type conflict that could arise would be if the name of a variable were identical to that of another one located elsewhere in the database. In this case, rename the second variable.

### Plug-in area variable

A plug-in area is always a Longint. There can never be a data type conflict.
For a plug-in area, the only possible data type conflict that could arise would be if the name of a variable were identical to that of another one located elsewhere in the database. In this case, rename the second variable.

### 4D Write Pro area variable

4D Write Pro areas are always Object type variables. There cannot be any typing conflict, unless the same variable name is used in another part of the application.

### Variables considered by default as Text

These variables are of the following types:

| |
|---|
| Non-enterable variable |
| Enterable variable |
| Drop-down list |
| Menu/drop-down list |
| Scrollable area |
| Combo box |
| Pop-up Menu |
| Tab control |
| Web area |
| Column of (array type) list box. |

These variables are divided into two categories:
- simple variables (enterable and non-enterable variables)
- display variables (drop-down lists, menus/drop-down lists, scrollable areas, pop-up menus, combo boxes, tab controls and columns of list boxes).

- *Simple variables*
  Their default data type is Text. When used in methods or object methods, they are assigned the data type selected by you. There is no danger of conflict other than one resulting from assigning the same name to another variable.

- *Display variables*
  Some variables are used to display arrays in forms. If default values have been entered in the Form editor, you must

explicitly declare the corresponding variables using the Array Declaration commands (**ARRAY BOOLEAN**, **ARRAY TEXT**, etc.).

### List boxes

Each list box adds several variables in the forms. The default type of these variables depends on the type of list box:

| | Array type list boxes | Selection type list boxes |
|---|---|---|
| List box | Boolean array | Number (not used) |
| Column of list box | Text array | Dynamic typing |
| Header | Number | Number |
| Footer | Dynamic typing | Dynamic typing |
| Row control array | Boolean array (Longint array accepted) | - |
| Styles | Longint array | Longint |
| Font colors | Longint array | Longint |
| Background colors | Longint array | Longint |

Make sure that you identify and type these variables and arrays correctly in order to avoid generating conflicts during compilation.

## Pointers

When you use pointers in your database, you take advantage of a powerful and versatile 4D tool. The compiler preserves all the benefits of pointers.

A pointer can point to variables of different data types. Do not create a conflict by assigning different data types to a variable. Be careful not to change the data type of a variable to which a pointer refers.

Here is an example of this problem:

```
Variable:=5.3
Pointer:=->Variable
Pointer->:=6.4
Pointer->:=False
```

In this case, your dereferenced pointer is a Real variable. By assigning it a Boolean value, you create a data type conflict.

If you need to use pointers for different purposes in the same method, make sure that your pointers are defined:

```
Variable:=5.3
Pointer:=->Variable
Pointer->:=6.4
Bool:=True
Pointer:=->Bool
Pointer->:=False
```

A pointer is always defined in relation to the object to which it refers. That is why the compiler cannot detect data type conflicts created by pointers. In case of a conflict, you will get no error message while you are in the typing phase or in the compilation phase.

This does not mean that the compiler has no way to detect conflicts involving pointers. The compiler can verify your use of pointers when you check the **Range Checking** option in the compilation Preferences (see the Design Reference manual).

## Plug-in Commands

### General points

During compilation, the compiler analyzes the definitions of the plug-in commands used in the database, i.e. the number and type of parameters of these commands. There is no danger of confusion at the typing level if your calls are consistent with the declaration of the method.

Make sure that your plug-ins are installed in the *PlugIns* folder, in one of the locations authorized by 4D: next to the database structure file or next to the executable application (Windows) / in the software package (Mac OS). For compatibility reasons, it is still possible to use the *Win4DX* or *Mac4DX* folder next to the structure file. For more information, refer to the Installation Guide of 4D.

The compiler does not duplicate these files, but analyzes them to determine the proper declaration of their routines.
If your plug-ins are located elsewhere, the compiler will ask you to locate them during typing, via an Open file dialog box.

### Plug-in commands receiving implicit parameters

Certain plug-ins, for example 4D Write, implement commands that implicitly call 4D commands.

Take the example of 4D Write. The syntax for the **WR ON EVENT** command is:
***WR ON EVENT****(area;event;eventMethod)*

The last parameter is the name of the method that you have created in 4D. This method is called by 4D Write each time the event is received. It automatically receives the following parameters:

| Parameters | Type | Description |
|---|---|---|
| $0 | Longint | Function return |
| $1 | Longint | 4D Write area |
| $2 | Longint | Shift key |
| $3 | Longint | Alt key (Windows); Option key (Mac OS) |
| $4 | Longint | Ctrl key (Windows), Command key (Mac OS) |
| $5 | Longint | Type of event |
| $6 | Longint | Value depends on the Event parameter |

For the compiler to take these parameters into account, you must make sure that they have been typed, either by a compiler directive, or by their usage in the method. If they have been used procedurally, the usage has to be explicit enough to be able to deduce the type clearly.

## 4D components

4D can be used to create and work with components. A 4D component is a set of 4D objects representing one or more functionalities and grouped in a structure file (called the matrix database), that can be installed in different databases (called host databases).

A host database running in interpreted mode can use either interpreted or compiled components indifferently. It is possible to install both interpreted and compiled components in the same host database. On the other hand, a host database running in compiled mode cannot use interpreted components. In this case, only compiled components can be used.

An interpreted host database containing interpreted components can be compiled if it does not call any methods of the interpreted component. If this is not the case, a warning dialog box appears when you attempt to compile the application and compilation is not possible.

A naming conflict can occur when a shared project method of the component has the same name as a project method of the host database. In this case, when the code is executed in the context of the host database, the method of the host database is called. This means that it is possible to “mask” the method of the component with a custom method (for example, to obtain a different functionality). When the code is executed in the context of the component, the method of the component is called. This masking will be indicated by a warning in the event of compilation of the host database.

If two components share methods having the same name, an error is generated when the host database is compiled.

For more information about components, please refer to the Design Reference manual.

## Handling local variables $0…$N and parameter passing

The handling of local variables follows all the rules that have already been stated. As with all other variables, their data types cannot be altered while the method executes. In this section, we examine two instances that could lead to data type conflicts:

- When you actually require retyping. The use of pointers helps avoid data type conflicts.
- When you need to address parameters by indirection.

### Using pointers to avoid retyping

A variable cannot be retyped. However, it is possible to use a pointer to refer to variables of different data types.

As an example, consider a function that returns the memory size of a one-dimensional array. In all but two cases, the result is a Real; for Text arrays and Picture arrays, the memory size depends on values that cannot be expressed numerically (see the **Arrays and Memory** section).

For Text and Picture arrays, the result is returned as a string of characters. This function requires a parameter: a pointer to the array whose memory size we want to know.

There are two methods to carry out this operation:

- Work with local variables without worrying about their data types; in such case, the method runs only in interpreted mode.
- Use pointers, and proceed in interpreted or in compiled mode.

**MemSize function, only in interpreted mode (example for Macintosh)**

```
$Size:=Size of array($1->)
$Type:=Type($1->)
Case of
   :($Type=Real array)
      $0:=8+($Size*10) ` $0 is a Real
   :($Type=Integer array)
      $0:=8+($Size*2)
   :($Type=LongInt array)
      $0:=8+($Size*4)
   :($Type=Date array)
      $0:=8+($Size*6)
   :($Type=Text array)
      $0:=String(8+($Size*4))+("+Sum of text lengths") ` $0 is a Text
   :($Type=Picture array)
      $0:=String(8+($Size*4))+("+Sum of picture sizes") ` $0 is a Text
   :($Type=Pointer array)
      $0:=8+($Size*16)
   :($Type=Boolean array)
      $0:=8+($Size/8)
End case
```

In the above method, the data type of $0 changes according to the value of $1; therefore, it is not compatible with the compiler.

**MemSize function in interpreted and compiled modes (example for Macintosh)**

Here, the method is written using pointers:

```
$Size:=Size of array($1->)
$Type:=Type($1->)
VarNum:=0
Case of
   :($Type=Real array)
      VarNum:=8+($Size*10) ` VarNum is a Real
   :($Type=Integer array)
      VarNum:=8+($Size*2)
   :($Type=LongInt array)
      VarNum:=8+($Size*4)
   :($Type=Date array)
      VarNum:=8+($Size*6)
   :($Type=Text array)
      VarText:=String(8+($Size*4))+("+Sum of text lengths")
   :($Type=Picture array)
      VarText:=String(8+($Size*4))+("+Sum of picture sizes")
   :($Type=Pointer array)
      VarNum:=8+($Size*16)
   :($Type=Boolean array)
      VarNum:=8+($Size/8)
End case
If(VarNum#0)
   $0:=->VarNum
Else
   $0:=->VarText
End if
```

Here are the key differences between the two functions:

- In the first case, the function's result is the expected variable,
- In the second case, the function's result is a pointer to that variable. You simply dereference your result.

### Parameter indirection

The compiler manages the power and versatility of parameter indirection. In interpreted mode, 4D gives you a free hand with numbers and data types of parameters. You retain this freedom in compiled mode, provided that you do not introduce data type conflicts and that you do not use more parameters than you passed in the calling method.

To prevent possible conflicts, parameters addressed by indirection must all be of the same data type.

This indirection is best managed if you respect the following convention: if only some of the parameters are addressed by indirection, they should be passed after the others.

Within the method, an indirection address is formatted: ${$i}, where $i is a numeric variable. ${$i} is called a generic parameter.

As an example, consider a function that adds values and returns the sum formatted according to a format that is passed as a parameter. Each time this method is called, the number of values to be added may vary. We must pass the values as parameters to the method and the format in the form of a character string. The number of values can vary from call to call.

This function is called in the following manner:

```
Result:=MySum("##0.00";125,2;33,5;24)
```

In this case, the calling method will get the string “182.70”, which is the sum of the numbers, formatted as specified. The function's parameters must be passed in the correct order: first the format and then the values.

Here is the function, named MySum:

```
$Sum:=0
For($i;2;Count parameters)
   $Sum:=$Sum+${$i}
End for
$0:=String($Sum;$1)
```

This function can now be called in various ways:

```
Result:=MySum("##0.00";125,2;33,5;24)
Result:=MySum("000";1;18;4;23;17)
```

As with other local variables, it is not necessary to declare generic parameters by compiler directive. When required (in cases of ambiguity or for optimization), it is done using the following syntax:

```
C_LONGINT(${4})
```

This command means that all parameters starting from the fourth (included) will be addressed by indirection and will be of the data type Longint. $1, $2 and $3 can be of any data type. However, if you use $2 by indirection, the data type used will be the generic type. Thus, it will be of the data type Longint, even if for you it was, for instance, of the data type Real.

**Note**: The compiler uses this command in the typing phase. The number in the declaration has to be a constant and not a variable.

## Reserved variables and constants

Some 4D variables and constants are assigned a data type and an identity by the compiler. Therefore, you cannot create a new variable, method, function or plug-in command using any of these variables or constant names. You can test their values and use them as you do in interpreted mode.

### System variables

Here is a complete list of 4D **System Variables** with their data types.

| Variable | Type |
|---|---|
| OK | Longint |
| Document | Text |
| FldDelimit | Longint |
| RecDelimit | Longint |
| Error | Longint |
| Error method | Text |
| Error line | Longint |
| Error formula | Text |
| MouseDown | Longint |
| KeyCode | Longint |
| Modifiers | Longint |
| MouseX | Longint |
| MouseY | Longint |
| MouseProc | Longint |

### Quick report variables

When you create a calculated column in a report, 4D automatically creates a variable C1 for the first one, C2 for the second one, C3 and so forth. This is done transparently.
If you use these variables in methods, keep in mind that, like other variables, C1, C2, ... Cn cannot be retyped.

### 4D predefined constants

A complete list of the predefined constants in 4D can be found using the **List of constant themes**. 4D constants are also displayed in the Explorer in Design mode.

## Syntax Details

The compiler expects that the usual syntactic rules for 4D commands are followed. It does not require any special modifications for databases that will be compiled.

This section nevertheless provides certain reminders and specific details:

- Some commands that affect a variable's data type may lead, if you are not careful, to data type conflicts.
- Since certain commands use more than one kind of syntax or parameters, it is to your advantage to know which is the most appropriate one to select.

### Strings

#### Character code(character)

For commands operating on strings, only the **Character code** function requires special attention. In interpreted mode, you can pass either a non-empty string or an empty string to this function.
In compiled mode, you cannot pass an empty string.
If you pass an empty string, and if the argument passed to **Character code** is a variable, the compiler will not be able to detect an error in compilation.

### Communications

**SEND VARIABLE(variable)**
**RECEIVE VARIABLE(variable)**
These two commands are used for writing and receiving variables sent to disk. Variables are passed as parameters to these commands.

The parameter you pass must always be of the same data type. Suppose you want to send a list of variables to a file. To eliminate the risk of changing data types inadvertently, we recommend that you specify the data type of the variables being sent at the head of the list. This way, when you receive these variables, you will always begin by getting an indicator. Then, when you call **RECEIVE VARIABLE**, the transfer is managed by a *Case of* statement.

Example:

```
SET CHANNEL(12;"File")
If(OK=1)
   $Type:=Type([Client]Total_TO)
   SEND VARIABLE($Type)
   For($i;1;Records in selection)
      $Send_TO:=[Client]Total_TO
      SEND VARIABLE($Send_TO)
   End for
End if
SET CHANNEL(11)
SET CHANNEL(13;"MyFile")
If(OK=1)
   RECEIVE VARIABLE($Type)
   Case of
      :($Type=Is string var)
         RECEIVE VARIABLE($String)
    `Processing variable received
      :($Type=Is real)
         RECEIVE VARIABLE($Real)
    `Processing variable received
      :($Type=Is text)
         RECEIVE VARIABLE($Text)
    `Processing variable received
   End case
```

```
End if
SET CHANNEL(11)
```

## Structure access

**Field (field pointer)** or **(table number;field number)**
**Table(table pointer)** or **(table number)** or **(field pointer)**
These two commands return results of different data types, according to the parameters passed to them:

- If you pass a pointer to the **Table** function, the result returned is a number.
- If you pass a number to the **Table** function, the result returned is a pointer.

## Documents

Keep in mind that the document references returned by the **Open document**, **Append document** and **Create document** functions are of the data type Time.

## Math

**Mod (value;divider)**
The expression “25 modulo 3” can be written in two different ways in 4D:

```
Variable:=Mod(25;3)
```

or

```
Variable:=25%3
```

The compiler sees a difference between the two: **Mod** applies to all numerics, while the operator % applies only to Integers and Long Integers. If the operand of the % operator exceeds the range of the Long Integer data type, the returned result is likely to be wrong.

## Exceptions

**IDLE**
**ON EVENT CALL (Method{; ProcessName})**
**ABORT**

### ON EVENT CALL

The **IDLE** command has been added to 4D language to manage exceptions. This command should be used whenever you use the **ON EVENT CALL** command.

This command could be defined as an event management directive.
Only the kernel of 4D is able to detect a system event (mouse click, keyboard activity, and so forth). In most cases, kernel calls are initiated by the compiled code itself, in a way that is transparent to the user.

On the other hand, when 4D is waiting passively for an event--for example, in a waiting loop--it is clear that there will be no call.

***Example under Windows***

```
 `MouseClick Method
If(MouseDown=1)
   <>vTest:=True
   ALERT("Somebody clicked the mouse")
End if

 `Wait Method
<>vTest:=False
ON EVENT CALL("MouseClick")
While(<>vTest=False)
 `Event's waiting loop
End while
ON EVENT CALL("")
```

In this case, you would add the **IDLE** command in the following manner:

```
 `Wait Method
<>vTest:=False
ON EVENT CALL("MouseClick")
While(<>vTest=False)
   IDLE
 `Kernel call to sense an event
End while
ON EVENT CALL("")
```

### ABORT

Use this command only in error-handling project methods. It works exactly as it does in 4D, except in a method that has been called by one of the following commands: **EXECUTE FORMULA**, **APPLY TO SELECTION** or **_o_APPLY TO SUBSELECTION**. Try to avoid this situation.

## Arrays

Seven 4D commands are used by the compiler to determine the data type of an array. They are:

**COPY ARRAY(source;destination)**
**SELECTION TO ARRAY(field;array)**
**ARRAY TO SELECTION(array;field)**
**SELECTION RANGE TO ARRAY(start;end;field;array)**
**LIST TO ARRAY(list;array{; itemRefs})**
**ARRAY TO LIST(array;list{; itemRefs})**
**DISTINCT VALUES(field;array)**

### COPY ARRAY

The **COPY ARRAY** command accepts two array type parameters. If one of the array parameters is not declared elsewhere, the compiler determines the data type of the undeclared array from the data type of the declared one.
This deduction is performed in the two following cases:

- The array typed is the first parameter. The compiler assigns the data type of the first array to the second array.
- The declared array is the second parameter. Here, the compiler assigns the data type of the second array to the first array.

Since the compiler is strict about data types, **COPY ARRAY** can be performed only from an array of a certain data type to an array of the same type.
Consequently, if you want to copy an array of elements whose data types are similar, i.e., Integers, Long Integers and Reals, or Texts and Strings, or Strings with different lengths, you have to copy the elements one by one.

Suppose you want to copy elements from an Integer array to a Real array. You can proceed as follows:

```
$Size:=Size of array(ArrInt)
ARRAY REAL(ArrReal;$Size)
 `Set same size for Real array as the Integer array
For($i;1;$Size)
   ArrReal{$i}:=ArrInt{$i}
 `Copy each of the elements
End for
```

Remember that you cannot change the number of dimensions of an array during the process. If you copy a one-dimensional array into a two-dimensional array, the compiler generates an error message.

### SELECTION TO ARRAY, ARRAY TO SELECTION, DISTINCT VALUES, SELECTION RANGE TO ARRAY

As with 4D in interpreted mode, these four commands do not require the declaration of arrays. The undeclared array will be assigned the data type of the field specified in the command.

If you write:

```
SELECTION TO ARRAY([MyTable]IntField;MyArray)
```

the data type of MyArray would be an Integer array having one dimension (assuming that IntField is an integer field).

If the array has been declared, make sure that the field is of the same data type. Although Integer, Longint and Real are similar types, they are not equivalent.

On the other hand, in the case of Text and String data types, you have a little more latitude. By default, if an array was not previously declared and you apply a command that includes a String type field as a parameter, the default data type assigned to the array is Text. If the array was previously declared as String or Text, these commands will follow your directives.

The same is true for Text type fields--your directives have priority.

Remember that the **SELECTION TO ARRAY**, **SELECTION RANGE TO ARRAY**, **ARRAY TO SELECTION** and **DISTINCT VALUES** commands can only be used with a one-dimensional array.

The **SELECTION TO ARRAY** command also has a second syntax:
**SELECTION TO ARRAY**(table;array).

In this case, the MyArray variable will be an array of Longints. The **SELECTION RANGE TO ARRAY** command works in the same way.

### LIST TO ARRAY, ARRAY TO LIST

The **LIST TO ARRAY** and **ARRAY TO LIST** commands only concern two types of arrays:

- one-dimensional String arrays, and
- one-dimensional Text arrays.

### Using pointers in array-related commands

The compiler cannot detect a data type conflict if you use a dereferenced pointer as a parameter to an array-declaration command. If you write:

```
SELECTION TO ARRAY([Table]Field:Pointer->)
```

where Pointer-> stands for an array, the compiler cannot check whether the field type and array type are identical. It is up to you to prevent such conflicts; you should type the array referred to by the pointer.

The compiler issues a warning whenever it encounters an array declaration statement in which one of the parameters is a pointer. These messages can be helpful in detecting this type of conflict.

### Local arrays

If your database uses local arrays (arrays recognized only in the method in which they were created), it is necessary to declare them explicitly in 4D before using them.

To declare a local array, use one of the array commands such as **ARRAY REAL**, **ARRAY INTEGER**, etc.

For example, if a method creates a local Integer array with 10 elements, you need to declare the array before using it. Use the command:

```
ARRAY INTEGER($MyArray;10)
```

## Language

**Get pointer(varName)**
**Type (object)**
**EXECUTE FORMULA(statement)**
**TRACE**
**NO TRACE**

### Get pointer

**Get pointer** is a function that returns a pointer to the parameter that you passed to it. Suppose you want to initialize an array of pointers. Each element in that array points to a given variable. Suppose there are twelve such variables named V1, V2, ···V12. You could write:

```
ARRAY POINTER(Arr;12)
Arr{1}:=->V1
Arr{2}:=->V2

Arr{12}:=->V12
```

You could also write:

```
ARRAY POINTER(Arr;12)
For($i;1;12)
   Arr{$i}:=Get pointer("V"+String($i))
End for
```

At the end of this operation, you get an array of pointers where each element points to a variable Vi.

These two sequences can be compiled. However, if the variables V1 to V12 are not used explicitly elsewhere in the database, the compiler cannot type them. Therefore, they must be used or declared explicitly elsewhere.
Such explicit declaration may be performed in two ways:

- By declaring V1, V2, ···V12 through a compiler directive:

```
C_LONGINT(V1;V2;V3;V4;V5;V6;V7;V8;V9;V10;V11;V12)
```

- By assigning these variables in a method:

```
V1:=0
V2:=0

V12:=0
```

### Type (object)

Since each variable in a compiled database has only one data type, the **Type** function may seem to be of no use. However, it can be useful when you work with pointers. For example, you may need to know the data type of the variable to which a pointer refers; due to the flexibility of pointers, one cannot always be sure to what object it points.

### EXECUTE FORMULA

This command offers benefits in interpreted mode that are not carried over to compiled mode.

In compiled mode, a method name passed as a parameter to this command is interpreted. Therefore, you miss some of the advantages provided by the compiler, and your parameter's syntax cannot be checked.

Moreover, you cannot pass local variables as parameters to it.

**EXECUTE FORMULA** can be replaced by a series of statements. Two examples are given below.

Given the following sequence:

```
i:=FormFunc
EXECUTE FORMULA("FORM SET INPUT (Form"+String(i)+")")
```

It can be replaced by:

```
i:=FormFunc
VarForm:="Form"+String(i)
FORM SET INPUT(VarForm)
```

Below is another example:

```
$Num:=SelPrinter
EXECUTE FORMULA("Print"+$Num)
```

Here, **EXECUTE FORMULA** can be replaced with *Case of*:

```
Case of
   :($Num=1)
      Print1
   :($Num=2)
      Print2
   :($Num=3)
      Print3
End case
```

The **EXECUTE FORMULA** command can always be replaced. Since the method to be executed is chosen from the list of the database's project methods or the 4D commands, there is a finite number of methods. Consequently, it is always possible to replace **EXECUTE FORMULA** with either *Case of* or with another command. Furthermore, your code will execute faster.

### TRACE, NO TRACE

These two commands are used in the debugging process. They serve no purpose in a compiled database. However, you can keep them in your methods; they will simply be ignored by the compiler.

## Variables

**Undefined(variable)**
**SAVE VARIABLES(document;variable1{; variable2…})**
**LOAD VARIABLES(document;variable1{; variable2…})**
**CLEAR VARIABLE(variable)**

### Undefined

Considering the typing process carried out by the compiler, a variable can never be undefined in compiled mode. In fact, all the variables have been defined by the time compilation has been completed. The **Undefined** function therefore always returns **False**, whatever parameter it is passed.

**Note:** To know if your application is running in compiled mode, call the **Is compiled mode** command.

### SAVE VARIABLES, LOAD VARIABLES

In interpreted mode, you can check that the document exists by testing if one of the variables is undefined after performing a **LOAD VARIABLES**. This is no longer feasible in compiled databases, because the **Undefined** function always returns **False**.

This test can be performed in either interpreted or compiled mode by:

1. Initializing the variables that you will receive to a value that is not a legal value for any of the variables.
2. Comparing one of the received variables to the initialization value after **LOAD VARIABLES**.

The method can be written as follows:

```
Var1:="xxxxxx"
 `"xxxxxx" is a value that cannot be returned by LOAD VARIABLES
Var2:="xxxxxx"
Var3:="xxxxxx"
Var4:="xxxxxx"
LOAD VARIABLES("Document";Var1;Var2;Var3;Var4)
If(Var1="xxxxxx")
 `Document not found

Else
 `Document found

End if
```

### CLEAR VARIABLE

This routine uses two different syntaxes in interpreted mode:

**CLEAR VARIABLE**(variable)

**CLEAR VARIABLE**("a")

In compiled mode, the first syntax of **CLEAR VARIABLE**(variable) reinitializes the variable (set to null for a numeric; empty string for a character string or a text, etc.), since no variable can be undefined in compiled mode.

Consequently, **CLEAR VARIABLE** does not free any memory in compiled mode, except in four cases: Text, Picture, BLOB and Array type variables.

For an array, **CLEAR VARIABLE** has the same effect as a new array declaration where the size is set to null.

For an array MyArray whose elements are of the *Integer* type, **CLEAR VARIABLE**(MyArray) has the same effect as one of the following expressions:

```
ARRAY INTEGER(MyArray;0)
 `if it as a one-dimensional array
ARRAY INTEGER(MyArray;0;0)
 `if it is a two-dimensional array
```

The second syntax, **CLEAR VARIABLE**("a"), is incompatible with the compiler, since compilers access variables by address, not by name.

## Pointers with certain commands

### Pointers with certain commands

The following commands have one common feature: they accept an optional first parameter [Table], and the second parameter can be a pointer.

| | |
|---|---|
| ADD TO SET | LOAD SET |
| APPLY TO SELECTION | LOCKED BY |
| COPY NAMED SELECTION | ORDER BY |
| CREATE EMPTY SET | ORDER BY FORMULA |
| CREATE SET | FORM SET OUTPUT |
| CUT NAMED SELECTION | PAGE SETUP |
| DIALOG | Print form |
| EXPORT DIF | PRINT LABEL |
| EXPORT SYLK | QR REPORT |
| EXPORT TEXT | QUERY |
| GOTO RECORD | QUERY BY FORMULA |
| GOTO SELECTED RECORD | QUERY SELECTION |
| _o_GRAPH TABLE | QUERY SELECTION BY FORMULA |
| IMPORT DIF | REDUCE SELECTION |
| IMPORT SYLK | RELATE MANY |
| IMPORT TEXT | REMOVE FROM SET |
| FORM SET INPUT | |

In compiled mode, it is easy to return the optional [Table] parameter. However, when the first parameter passed to one of these commands is a pointer, the compiler does not know to what the pointer is referring; the compiler treats it as a table pointer.

Take the case of the **QUERY** command whose syntax is as follows:
**QUERY**({table{;formula{;*}}})

The first element of the *formula* parameter must be a field.

If you write :

```
QUERY(PtrField->=True)
```

the compiler will look for a symbol representing a field in the second element. When it finds the "=" sign, it will issue an error message, since it cannot identify the command with an expression that it knows how to process.

On the other hand, if you write:

```
QUERY(PtrTable->;PtrField->=True)
```

or

```
QUERY([Table];PtrField->=True)
```

you will avoid any possible ambiguity.

### Direct use of commands returning pointers

When using pointers, there is a particularity concerning commands where the first parameter [aTable] and second parameter are both optional. In this context, for internal reasons, the compiler does not allow a command that returns a pointer (for example **Current form table**) to be passed directly as a parameter (an error is generated).

This is the case, for example, of the **FORM SCREENSHOT** command. The following code works in interpreted mode but is rejected during compilation:

```
 //triggers a compilation error
FORM SCREENSHOT(Current form table->;$formName;$myPict)
```

In this case, you can just use an intermediate variable in order for this code to be validated by the compiler:

```
 //equivalent compilable code
C_POINTER($ptr)
$ptr:=Current form table
FORM SCREENSHOT($ptr->;$formName;$myPict)
```

## Constants

If you create your own 4DK# resources (constants), make sure that numerics are declared as Longints (L) or Reals (R) and character strings as Strings (S). Any other type will generate a warning.

# Optimization Hints

It is difficult to state a definitive “good-programming” method, but we wish to stress the advantages of well-structured programs. The capacity for structured programming in 4D can be a great help.

The compilation of a well-structured database can yield much better results than the same effort performed in a poorly-designed one. For instance, if you write a generic method to manage n object methods, you will get higher quality results in both interpreted and compiled modes than in a situation where n object methods comprise n times the same set of statements.

In other words, the quality of the programming does have an impact on the quality of the compiled code.

With practice, you can gradually improve your 4D code. Frequent use of the compiler gives you corrective feedback that enables you to reach instinctively for the most efficient solution.
In the meantime, we can offer some advice and a few tricks that will save you time in performing simple, recurring tasks.

## Using comments in code

Certain programming techniques may make your code less readable both for yourself or another person at a later time. Because of this, we encourage you to comment your methods with a lot of detail. In fact, while excessive comments have a tendency to slow down interpreted databases, they have absolutely no influence on the execution time in a compiled database.

## Using compiler directives to optimize code

Compiler directives can help you speed up your code considerably. When typing variables on the basis of their use, the compiler uses the data type with the largest scope possible so as not to penalize you. For example, if you do not type the variable defined by the statement: *Var:= 5*, the compiler will type it as Real, even if it could be declared an Integer.

### Numeric Variables

The compiler gives numeric variables (not typed by compiler directives) the default data type Real if the Database Settings are not set to anything else. But calculations performed on a Real are slower than on a Longint. If you know that a numeric variable will always be an integer, it is to your advantage to declare it through the compiler directive **C_LONGINT**.

For example, it is good practice to declare your loop counters as Integers.

Some 4D functions return Integers (e.g., **Character code**, **Int**, etc.). If you assign the result of one of these functions to an untyped variable of your database, the compiler types it as Real rather than as Integer. Remember to declare such variables with compiler directives whenever you are sure that they will not be used in a different context.

Here is a simple example of a function that returns a random value with a given range:

```
$0:=Mod(Random;($2-$1+1))+$1
```

It will always return an integer. Written this way, the compiler will type $0 as Real rather than Integer or Longint. It is preferable, therefore, to include a compiler directive in the method:

```
C_LONGINT($0)
$0:=Mod(Random;($2-$1+1))+$1
```

The parameter returned by the method will take less space in memory and will be much faster.

Here is another example. Suppose you typed two variables as Longint:

```
C_LONGINT($var1;$var2)
```

and a third non-typed variable receives the sum of the other two variables:

```
$var3:=$var1+$var2.
```

The compiler will type the third variable, $var3, as Real. You will have to explicitly declare it as Longint if you want the result to be a long integer.

**Note:** Be careful with the computation mode in 4D. In compiled mode, it is not the data type of the variable that receives the calculation which determines the computation mode, but rather the data types of the operands.

In the following example, the calculation is based on long integers:

```
C_REAL($var3)
C_LONGINT($var1;$var2)
$var1:=2147483647
$var2:=1
$var3:=$var1+$var2
```

$var3 is equal to −2147483648 in both compiled mode and interpreted mode.

However, in this example:

```
C_REAL($var3)
C_LONGINT($var1)
$var1:=2147483647
$var3:=$var1+1
```

for optimization reasons, the compiler considers the value 1 as an integer. In compiled mode, $var3 is equal to −2147483648 because the calculation is based on Longints. In interpreted mode, $var3 is equal to 2147483648 because the calculation is based on Reals.

Buttons are a specific case of a Real that can be declared as Longint.

### Strings

If you want to test the value of a character, make the comparison on its **Character code** value rather than on the character itself. The regular character comparison procedure considers all of the character's alternatives, such as diacritical marks.

## Various observations

### Two-dimensional arrays

The processing of two-dimensional arrays is better managed if the second dimension is larger than the first.

For example, an array declared as:

```
ARRAY INTEGER(Array;5;1000)
```

will be better managed than an array declared as:

```
ARRAY INTEGER(Array;1000;5)
```

### Fields

Whenever you need to perform several calculations on a field, you can improve performance by storing the value of that field in a variable and performing your calculations on the variable rather than the field. Consider the following method:

```
Case of
   :([Client]Dest="New York City")
      Transport:="Messenger"
   :([Client]Dest="Puerto Rico")
      Transport:="Air mail"
   :([Client]Dest="Overseas")
      Transport:="Express mail service"
   Else
      Transport:="Regular mail service"
End case
```

This method will take longer to execute than if it were written:

```
$Dest:=[Client]Dest
Case of
   :($Dest="New York City")
      Transport:="Messenger"
   :($Dest="Puerto Rico")
```

```
      Transport:="Air mail"
   :($Dest="Overseas")
      Transport:="Express mail service"
   Else
      Transport:="Regular mail service"
End case
```

### Pointers

As is the case with fields, it is faster to work with variables than with dereferenced pointers. Whenever you need to perform several calculations on a variable referenced by a pointer, you can save time by storing the dereferenced pointer in a variable.

For example, suppose you use a pointer, MyPtr, to refer to a field or to a variable. Then, you want to perform a set of tests on the value referenced by MyPtr. You could write:

```
Case of
   :(MyPtr->=1)
      Sequence 1
   :(MyPtr->=2)
      Sequence 2

End case
```

The set of tests would be performed faster if it were written:

```
Temp:=MyPtr->
Case of
   :(Temp=1)
      Sequence 1
   :(Temp=2)
      Sequence 2

End case
```

### Local variables

Use local variables wherever possible to structure you code. Using local variables has the following advantages:

- Local variables take less space when used in a database. They are created when the method in which they are used is entered, and they are destroyed when the method finishes executing.
- The code generated is optimized for local variables, especially for those of the type Longint. This is useful for counters in loops.

# Error messages

This section describes the different messages generated by the compiler. These messages are of several different types:

- warnings, that help you avoid common pitfalls;
- errors, that it is up to you to correct;
- range checking messages, generated within 4D.

## Warnings

These messages are generated throughout the compilation process. Each message is accompanied here with an example of the problem and, when necessary, an additional explanation.

Pointer in COPY ARRAY

```
COPY ARRAY(Pointer->;Array)
```

Pointer in SELECTION TO ARRAY

```
SELECTION TO ARRAY(Pointer->;MyArray)
SELECTION TO ARRAY([MyTable]MyField;Pointer->)
```

Pointer in ARRAY TO SELECTION

```
ARRAY TO SELECTION(Pointer->;[MyTable]MyField)
```

Pointer in LIST TO ARRAY

```
LIST TO ARRAY(List;Pointer->)
```

Pointer in ARRAY TO LIST

```
ARRAY TO LIST(Pointer->;List)
```

Pointer in an array declaration

```
ARRAY REAL(Pointer->;5)
```

The command **ARRAY REAL**(Array;Pointer->) does not generate this warning. The value of the dimension of an array does not have any influence on its type. In this example, the array referred to by the pointer must have been defined elsewhere.

Pointer in DISTINCT VALUES

```
DISTINCT VALUES(Pointer->;Array)
```

Using the function Undefined is not advised.

```
If(Undefined(Variable))
```

The **Undefined** function always returns *FALSE* in a compiled database.

This method is protected by a password.

An automatic action button does not have a name in the MyForm form on page X.
All of your buttons should have names to avoid conflicts.

Assumes that the pointer points to an alphanumeric expression.

```
Pointer->≤2≥:="a"
```

Assumes that the string index is numeric.

```
String≤Pointer->≥:="a"
```

Assumes that the array index is of type real.

```
ALERT(MyArray{Pointer->})
```

Missing parameter in the plug-in procedure call.

```
WR SET FONT(Area)
```

**Note:** You can enable and disable warnings individually using the following tags:
**//%W-warning_number** to disable a warning
**//%W+warning_number** to enable a warning
Enabling and disabling warnings in this way is effective for all the code parsed subsequently in the compilation plan. If you want to generally disable or enable warnings, you can just insert the appropriate tag in a method named "Compiler_xxx" since these methods are the first ones parsed by the compiler. For example, to disable the "Pointer in COPY ARRAY" warning, you can insert the "//%W-518.1" tag at the desired location.

## Error messages

These messages are generated throughout the compilation process. It is up to you to correct these errors in order to for the compiler to be able to generate a compiled database. Each message is accompanied here with an example of the problem and, when necessary, an additional explanation.

**Typing**

The type of the variable is not compatible with the operator. Cannot make an assignment with those types.

```
MyReal:=12.3
MyBoolean:=True
MyReal:=MyBoolean
```

Changing the number of dimensions of an array.

```
ARRAY TEXT(MyArray;5;5)
ARRAY TEXT(MyArray;5)
```

Typing conflict on the MyArray variable in the form.

```
ARRAY INTEGER(MyArray)
```

Declaring an array without dimensions.

```
ARRAY INTEGER(MyArray)
```

Variable expected.

```
COPY ARRAY(MyArray;"")
```

The type of Variable is unknown. This variable is used in the method M1.
The type of Variable cannot be determined. A compiler directive is necessary.

Invalid constant type

```
OK:="The weather is nice"
```

The method M1 is unknown.
The line contains a call to a method that does not exist or no longer exists.

Incorrect usage of a field.

```
MyDate:=Add to date(BooleanField;1;1;1)
```

The variable Variable is not a method.

```
Variable(1)
```

The variable Variable is not an array.

```
Variable{5}:=12
```

The result of the function is not compatible with the expression.

```
Text:="Number"+Num(i)
```

The types of the variables used in this expression are not compatible.

```
Integer:=MyDate*Text
```

Changing the type of the variable $i from type Fixed string to type Real.

```
$i:="3"
$($i):=5
```

The array index is not a number.

```
IntArray{"3"}:=4
```

Retyping the variable Variable from type Text to an array of type Text.

```
C_TEXT(Variable)
COPY ARRAY(TextArray;Variable)
```

Retyping of the variable Variable from type Text to type Real.

```
Variable:=Num(Variable)
```

Retyping the array MyBoolean from array of type Boolean to variable of type Real.

```
Variable:=MyBoolean
```

Retyping the array IntArray from array of type Integer to array of type Text.

```
ARRAY TEXT(IntArray;12)
```

if IntArray was declared elsewhere as an Integer array.

Trying to dereference a variable which is not of type Pointer.

```
Variable->:=5
```

if Variable is not of the type Pointer.

Retyping of the variable Var1 from type Text to type Number.

```
Var1:=3.5
```

Incorrect usage of a field.

```
Variable:=[MyTable]MyField
```

[MyTable]MyField is a Date field. Variable is of the type Number.

**Syntax**

The result of the function is not a pointer.

```
Variable:=Num("The weather is nice")->
```

It is not possible to dereference this function.

Syntax error.

```
If(Boolean)
End for
```

Too many opening curly brackets ({) .
The line contains more opening brackets than closing brackets.

Too many closing curly brackets (}).
The line contains more closing brackets than opening brackets.

Closing parenthesis ) expected.
The line contains more opening parentheses than closing parentheses.

Opening parenthesis ( expected.
The line contains more closing parentheses than opening parentheses.

Field expected.

```
If(Modified(Variable))
```

Opening curly bracket expected.

```
C_INTEGER($
```

Variable expected.

```
C_INTEGER([MyTable]MyField)
```

Constant number expected.

```
C_INTEGER($["3"])
```

Semicolon ; expected.

```
COPY ARRAY(Array1 Array2)
```

**Mac OS**

```
MyString≤3:="a"
```

Too many closing character reference symbols.

```
MyString3≥:="a"
```

**Windows**

```
MyString[[3:="a"
```

Too many closing character reference symbols.

```
MyString 3]]:="a"
```

Did not expect a subtable.

```
ARRAY TO SELECTION(Array;Subtable)
```

The argument of an IF statement must be a boolean.

```
If(Pointer)
```

Expression is too complex.
Divide your statement into several shorter statements.

Method is too complex.
Too many Case of...End case and/or If…End if structures.

Unknown field.
Your method, possibly copied from another database, contains •???• instead of a field name.

Unknown table.
Your method, possibly copied from another database, contains •???• instead of a table name.

Pointer to an incorrect expression.

```
Pointer:=->Variable+3
```

Incorrect usage of string index.

```
MyReal≤3≥or MyReal[[3]]
```

or

```
MyString≤Variable≥or MyString[[Variable]]
```

where Variable is not a Number variable.

**Parameters**

The result of this function cannot be passed as a parameter to this method or command.

```
MyMethod(Num(MyString))
```

if MyMethod expects a Boolean expression.

Too many parameters have been passed to this method.

```
DEFAULT TABLE(Table:Form)
```

This value cannot be passed as a parameter to this method or command.

```
MyMethod(3+2)
```

if MyMethod expects a Boolean expression.

Function result type conflict.

```
C_INTEGER($0)
$0:=False
```

Generic parameter type conflict.

```
C_INTEGER(${3})
For($i;3;5)
   ${$i}:=String($i)
End for
```

This 4D command does not require any parameters.

```
SHOW TOOL BAR(MyVar)
```

This 4D command requires at least one parameter.

```
DEFAULT TABLE
```

MyString cannot be passed as a parameter to that method.

```
MyMethod(MyString)
```

if MyMethod is expecting a Boolean parameter.

The type of the parameter $1 is different in the calling and in the called method.

```
Calculate("3+2")
```

with the directive **C_INTEGER**($1) in Calculate.

One of the parameters in COPY ARRAY is a variable.

```
COPY ARRAY(Variable;Array)
```

Retyping of the variable $1 from type Number to type Text.

```
$1:=String($1)
```

An array cannot be a parameter.

```
ReInit(MyArray)
```

To pass an array in a method, you need to pass a pointer to the array.

**Operators**

The type of the variable is not compatible with the operator.

```
Bool2:=Bool1+True
```

Addition cannot be performed on Boolean fields.

Did not expect the operator >.

```
QUERY(MyTable;[MyTable]MyField=0;>)
```

Cannot compare two variables of those types.

```
If(Number=Picture2)
```

Cannot negate this type of variable.

```
Boolean:=-False
```

**Plug-in Commands**

The plug-in command PExt does not seem to be correctly defined.

Not enough parameters were passed to this plug-in command.

Too many parameters were passed to this plug-in command.

The plug-in command Variable does not seem to be correctly defined.

**General Errors**

Two methods have the same name : Name.
To compile your database, all of the project methods must have different names.

Internal error # xx.
If this message appears, call 4D Technical Support and report the error number.

The variable Variable could not be typed. This variable is used in the method M1.
The Variable type cannot be determined. A compiler directive is necessary.

The original method is damaged.
The method is damaged in the original structure. Delete it or replace it.

Unknown 4D command.
The method is damaged.

Retyping the variable Variable in the form Form.
This message appears if you give, for example, the name OK to a variable of the type Graph in a form.

The name of the function Name is also the name of a variable in the form.
Rename either the function or the variable.

A method and a variable have the same name : Name.
Rename either the method or the variable.

A plug-in command and a variable have the same name : Name.
Rename either the plug-in command or the variable.

Cannot call a command that is not thread-safe from a method declared as thread-safe.
Modify the method so that it becomes thread-safe (do not use any command that is not thread-safe) or change the declaration property of the method in order to start a cooperative process

Cannot call 'MethodName' method that is not thread-safe from a method declared thread-safe.
Modify the method so that it becomes thread-safe or change the declaration property of the method in order to start a cooperative process

## Range-checking messages

These messages are generated in 4D while the compiled database is running. They are displayed in a specific error window.

The result is out of range.
If MyArray is a five-element array, this message appears if you try to access element 17 in the array.

Division by zero.

```
Var1:=0
Var2:=2
Var3:=Var2/Var1
```

Accessing a parameter that does not exist.
Using the $4 local variable when only three parameters have been passed to the current method.

The pointer is not properly initialized.

```
MyPointer->:=5
```

if MyPointer has not yet been initialized.

The destination string is smaller than the source.

```
C_STRING(MyString1;5)
C_STRING(MyString2;10)
MyString2:="Flowers"
MyString1:=MyString2
```

Invalid character reference.

```
i:=-30
MyString≤i≥:=MyString2 or MyString[[i]]:=MyString2
```

The parameter is an empty string.

```
MyString≤1≥:=""
MyString[[1]]:=""
```

Modulo by zero.

```
Var1:=0
Var2:=2
Var3:=Var2% Var1
```

Invalid parameters in an EXECUTE command.

```
EXECUTE FORMULA("MyMethod(MyAlpha)")
```

if MyMethod expects a parameter other than an Alphanumeric.

Pointer to an unknown variable.

```
MyPointer:=Get pointer("Variable")
MyPointer:="MyString"
```

if Variable does not appear explicitly in the database.

Attempting to retype by using a pointer.

```
Boolean:=Pointer->
```

if Pointer points to a field of type Integer.

Bad usage of a pointer or pointer to an unknown variable.

```
Character:=StringVar≤Pointer->≥
Character:=StringVar[[Pointer]]
```

if Pointer does not point to a Number.

# C_BLOB

C_BLOB ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Description

**C_BLOB** casts each specified variable as a BLOB variable.

The first form of the command, in which the optional *method* parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

**Note:** This form can be used in interpreted databases.

The second form of the command, in which the optional *method* parameter IS passed, is used to predeclare to the compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

**WARNING:** The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

**Advanced Tip:** The syntax **C_BLOB**(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration **C_BLOB**(${5}) tells 4D and the compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the **Count parameters** command.

## Examples

See examples in the **Compiler Commands** section.

# C_BOOLEAN

C_BOOLEAN ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Description

The **C_BOOLEAN** command casts each specified variable as a Boolean variable.

The first form of the command, in which the optional *method* parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

**Note:** This form can be used in interpreted databases.

The second form of the command, in which the optional *method* parameter IS passed, is used to predeclare to the compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

**WARNING:** The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

**Advanced Tip:** The syntax **C_BOOLEAN**(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration **C_BOOLEAN**(${5}) tells 4D and the compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the **Count parameters** command.

## Examples

See examples in the section **Compiler Commands**.

# C_DATE

C_DATE ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Description

The **C_DATE** command casts each specified variable as a Date variable.

The first form of the command, in which the optional *method* parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

**Note:** This form can be used in interpreted databases.

The second form of the command, in which the optional *method* parameter IS passed, is used to predeclare to the compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

**WARNING:** The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

**Advanced Tip:** The syntax **C_DATE**(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration **C_DATE**(${5}) tells 4D and the compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the **Count parameters** command.

## Examples

See examples in the section **Compiler Commands**.

# C_LONGINT

C_LONGINT ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Description

The **C_LONGINT** command casts each specified variable as a Long Integer variable.

The first form of the command, in which the optional *method* parameter is NOT passed, is used to declare and type any process, interprocess or local variable.

**Note:** This form can be used in interpreted databases.

The second form of the command, in which the optional *method* parameter IS passed, is used to predeclare to the compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

**WARNING:** The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

**Advanced Tip:** The syntax **C_LONGINT**(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration **C_LONGINT**(${5}) tells 4D and the compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the **Count parameters** command.

## Examples

See examples in the section **Compiler Commands**.

# C_OBJECT

C_OBJECT ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Name of method |
| variable | Variable | ⇒ | Name(s) of variable(s) or parameter(s) ${...} to declare |

## Description

The **C_OBJECT** command assigns the *Object* type to all the variables that are specified.

The *Object* type is supported by the 4D language starting with v14. Objects of this type are managed by the commands of the **Objects (Language)** theme.

You use the first syntax of the command (when the *method* parameter is not passed) to declare and type a process, interprocess or local variable. This syntax can be used in interpreted databases.

You use the second syntax of the command (when the *method* parameter is passed) to declare the method's result and/or parameters ($0, $1, $2, etc.) to the compiler in advance. You must use this syntax if you want to skip the variable typing phase in order to save time when the database is compiled.

**WARNING:** You cannot execute the second syntax in interpreted mode. For this reason, when you use this syntax, you should store it in a method (whose name must begin with "COMPILER") that is not executed in interpreted mode.

**Advanced use:** You can use the **C_OBJECT**(${...}) syntax to declare a variable number of parameters of the same type for a method as long as they are the last parameters of this method. For example, the declaration **C_OBJECT**(${5}) indicates to the compiler that beginning with the fifth parameter, the method can receive a variable number of parameters of this type.

## Example

Refer to the **Compiler Commands** section.

# C_PICTURE

C_PICTURE ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Description

The **C_PICTURE** command casts each specified variable as a Picture variable.

The first form of the command, in which the optional *method* parameter is NOT passed, is used to declare and type any process, interprocess or local variable.

**Note:** This form can be used in interpreted databases.

The second form of the command, in which the optional *method* parameter IS passed, is used to predeclare to the compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

**WARNING:** The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

**Advanced Tip:** The syntax **C_PICTURE**(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration **C_PICTURE**(${5}) tells 4D and the compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the **Count parameters** command.

## Examples

See examples in the section **Compiler Commands**.

# C_POINTER

C_POINTER ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Description

The **C_POINTER** command casts each specified variable as a Pointer variable.

The first form of the command, in which the optional *method* parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

**Note:** This form can be used in interpreted databases.

The second form of the command, in which the optional *method* parameter IS passed, is used to predeclare to the compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

**WARNING:** The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with "COMPILER."

**Advanced Tip:** The syntax **C_POINTER**(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration **C_POINTER**(${5}) tells 4D and the compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the **Count parameters** command.

## Examples

See examples in the section **Compiler Commands**.

# C_REAL

C_REAL ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Description

The **C_REAL** command casts each specified variable as a Real variable.

The first form of the command, in which the optional *method* parameter is NOT passed, is used to declare and type any process, interprocess or local variable.

**Note:** This form can be used in interpreted databases.

The second form of the command, in which the optional *method* parameter IS passed, is used to predeclare to the compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

**WARNING:** The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

**Advanced Tip:** The syntax **C_REAL**(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration **C_REAL**(${5}) tells 4D and the compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the **Count parameters** command.

## Examples

See examples in the section **Compiler Commands**.

# C_TEXT

C_TEXT ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Description

The **C_TEXT** command casts each specified variable as a Text variable.

The first form of the command, in which the optional *method* parameter is NOT passed, is used to declare and type any process, interprocess or local variable.

**Note:** This form can be used in interpreted databases.

The second form of the command, in which the optional *method* parameter IS passed, is used to predeclare to the compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

**WARNING:** The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

**Advanced Tip:** The syntax **C_TEXT**(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration **C_TEXT**(${5}) tells 4D and the compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the **Count parameters** command.

## Examples

See examples in the section **Compiler Commands**.

# C_TIME

C_TIME ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Description

The **C_TIME** command casts each specified variable as a Time variable.

The first form of the command, in which the optional *method* parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

**Note:** This form can be used in interpreted databases.

The second form of the command, in which the optional *method* parameter IS passed, is used to predeclare to the compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

**WARNING:** The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

**Advanced Tip:** The syntax **C_TIME**(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration **C_TIME**(${5}) tells 4D and the compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the **Count parameters** command.

## Examples

See examples in the section **Compiler Commands**.

# IDLE

IDLE
Does not require any parameters

## Description

The **IDLE** command is designed only for use with the compiler. This command is only used in compiled databases in which user-defined methods are written so that no calls are made back to the 4D engine. For example, if a method has a *For* loop in which no 4D commands are executed, the loop could not be interrupted by a process installed with **ON EVENT CALL**, nor could a user switch to another application. In this case, you should insert **IDLE** to allow 4D to trap events. If you do not want any interruptions, omit **IDLE**.

## Example

In the following example, the loop would never terminate in a compiled database without the call to **IDLE**:

```
  ` Do Something Project Method
ON EVENT CALL("EVENT METHOD")
◊vbWeStop:=False
MESSAGE("Processing..."+Char(13)+"Type any key to interrupt...")
Repeat
  ` Do some processing that doesn't involve a 4D command
   IDLE
Until(◊vbWeStop)
ON EVENT CALL("")
```

with:

```
  ` EVENT METHOD Project Method
If(Undefined(KeyCode))
   KeyCode:=0
End if
If(KeyCode#0)
   CONFIRM("Do you really want to stop this operation?")
   If(OK=1)
      ◊vbWeStop:=True
   End if
End if
```

# _o_C_GRAPH

_o_C_GRAPH ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | String | ⇒ | Name of method |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Compatibility note

Variables of the Graph area type are obsolete and no longer supported since 4D v14. You need to use picture variables (see **GRAPH**).

# _o_C_INTEGER

_o_C_INTEGER ( {method ;} variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ➡ | Optional name of method |
| variable | | ➡ | Name of variable(s) to declare |

## Preliminary Note

The **_o_C_INTEGER** command is still present in 4D for compatibility with old databases. In fact, 4D and the compiler retype Integers into Longints internally. For example:

```
_o_C_INTEGER($MyVar)
$TheType:=Type($MyVar) //$TheType = 9 (Is longint)
```

# _o_C_STRING

_o_C_STRING ( {method ;} size ; variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| method | Method | ⇒ | Optional name of method |
| size | Longint | ⇒ | Size of the string |
| variable | Variable | ⇒ | Name of variable(s) to declare |

## Compatibility note

The operation of the **_o_C_STRING** command is strictly identical to the **C_TEXT** command (the *size* parameter is ignored). It is now recommended to only use **C_TEXT** in your 4D developments.

# Data Entry

- ACCEPT
- ADD RECORD
- CANCEL
- DIALOG
- Modified
- MODIFY RECORD
- Old
- REJECT
- *_o_ADD SUBRECORD*
- *_o_MODIFY SUBRECORD*

# ACCEPT

ACCEPT
Does not require any parameters

## Description

The **ACCEPT** command is used in form or object methods (or in subroutines) to:

- accept a new or modified record or subrecord, for which data entry has been initiated using **ADD RECORD**, **MODIFY RECORD**, **_o_ADD SUBRECORD** or **_o_MODIFY SUBRECORD**.
- accept a form displayed with the **DIALOG** command.
- exit a form displaying a selection of records, using **DISPLAY SELECTION** or **MODIFY SELECTION**.

**ACCEPT** performs the same action as if a user had pressed the Enter key. After the form is accepted, the OK system variable is set to 1.

**ACCEPT** is commonly executed as a result of choosing a menu command. **ACCEPT** is also commonly used in the object method of a “no action” button.

It is also often used in the optional close box method for the **Open window** command. If there is a Control-menu box on a window, **ACCEPT** or **CANCEL** can be called, in the method to be executed, when the Control-menu box is double-clicked or the Close menu command is chosen.

**ACCEPT** cannot be queued up. In response to an event, executing two **ACCEPT** commands in a row from within a method would have the same effect as executing one.

# ADD RECORD

ADD RECORD ( {aTable}{;}{*} )

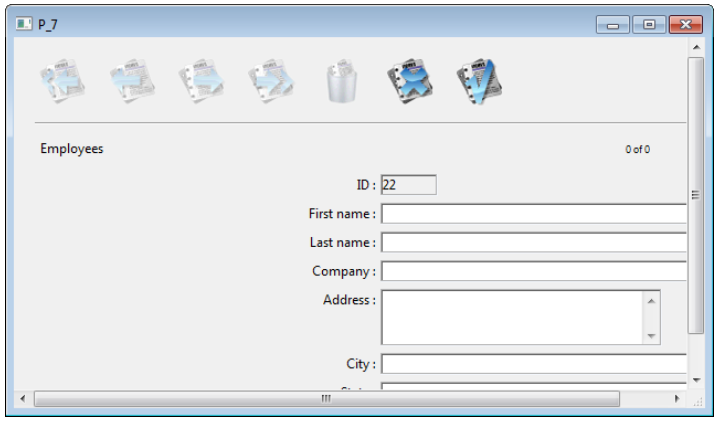
| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table to use for data entry, or Default table, if omitted |
| * | | ➡ | Hide scroll bars |

## Description

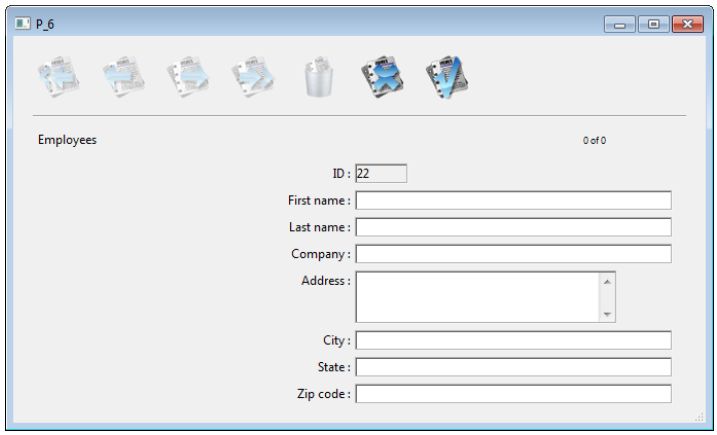
The **ADD RECORD** command lets the user add a new record to the database for the table *aTable* or for the default table, if you omit the *aTable* parameter.

**ADD RECORD** creates a new record, makes the new record the current record for the current process, and displays the current input form. In the Application environment, after the user has accepted the new record, the new record is the only record in the current selection.

The following figure shows a typical data entry form.




The form is displayed in the frontmost window of the process. The window has scroll bars and a size box. Specifying the optional * parameter causes the window to be drawn without scroll bars and the form window can no longer be reduced:




**ADD RECORD** displays the form until the user accepts or cancels the record. If the user is adding several records, the command must be executed once for each new record.

The record is saved (accepted) if the user clicks an Accept button or presses the Enter key (numeric keypad), or if the **ACCEPT** command is executed.

The record is not saved (canceled) if the user clicks a Cancel button or presses the cancel key combination (Ctrl-Period on Windows, Command-Period on Macintosh), or if the **CANCEL** command is executed.

**Note:** This command does not require *aTable* to be in read/write mode. It can be used even when the table is in read-only mode (see **Record Locking**).

After a call to **ADD RECORD**, OK is set to 1 if the record is accepted, to 0 if canceled.

**Note:** Even when canceled, the record remains in memory and can be saved if **SAVE RECORD** is executed before the current record pointer is changed.

## Example 1

The following example is a loop commonly used to add new records to a database:

```
FORM SET INPUT([Customers];"Std Input") ` Set input form for [Customers] table
Repeat ` Loop until the user cancels
   ADD RECORD([Customers];*) ` Add a record to the [Customers] table
Until(OK=0) ` Until the user cancels
```

## Example 2

The following example queries the database for a customer. Depending on the results of the search, one of two things may happen. If no customer is found, then the user is allowed to add a new customer with **ADD RECORD**. If at least one customer is found, the user is presented with the first record found, which can be modified with **MODIFY RECORD**:

```
READ WRITE([Customers])
FORM SET INPUT([Customers];"Input") ` Set the input form
vlCustNum:=Num(Request("Enter Customer Number:")) ` Get the customer number
If(OK=1)
   QUERY([Customers];[Customers]CustNo=vlCustNum) ` Look for the customer
   If(Records in selection([Customers])=0) ` If no customer is found…
      ADD RECORD([Customers]) ` Add a new customer
   Else
      If(Not(Locked([Customers])))
         MODIFY RECORD([Customers]) ` Modify the record
         UNLOAD RECORD([Customers])
      Else
         ALERT("The record is currently being used.")
      End if
   End if
End if
```

## System variables and sets

Accepting the record sets the OK system variable to 1; canceling it sets the OK system variable to 0. The OK system variable is set only after the record is accepted or canceled.

# CANCEL

CANCEL
Does not require any parameters

## Description

The **CANCEL** command is used in form or object methods (or in a subroutine) to:

- cancel a new or modified record or subrecord, for which data entry has been initiated using **ADD RECORD**, **MODIFY RECORD**, **_o_ADD SUBRECORD** or **_o_MODIFY SUBRECORD**.
- cancel a form displayed with the **DIALOG** command.
- exit a form displaying a selection of records, using **DISPLAY SELECTION** or **MODIFY SELECTION**.
- cancel the printing of a form that is about to be printed using the **Print form** command (see below).

In the context of data entry, **CANCEL** performs the same action as if the user had pressed the cancel key (**Esc**).

**CANCEL** is commonly executed as a result of a menu command being chosen. **CANCEL** is also commonly used in the object method of a “no action” button.

It is also often used in the optional close box method for the **Open window** command. If there is a Control-menu box on a window, **ACCEPT** or **CANCEL** can be called, in the method to be executed, when the Control-menu box is double-clicked or the **Close** menu command is chosen.

**CANCEL** cannot be queued up. Executing two **CANCEL** commands in a row from within a method in response to an event would have the same effect as executing only one.

Finally, this command can be used in the On Printing Detail form event, when using the **Print form** command. In this context, the **CANCEL** command suspends the printing of the form that is about to be printed, then resumes it on the next page. This mechanism can be used to manage form printing when there is a lack of space or if a page break is required.

**Note:** This operation differs from that of the **PAGE BREAK**(*) command that cancels ALL the forms waiting to be printed.

## Example

Refer to the example of the **SET PRINT MARKER** command.

## System variables and sets

When the **CANCEL** command is executed (form or printing cancelled), the system variable OK is set to 0.

# DIALOG

DIALOG ( {aTable ;} form {; *} )

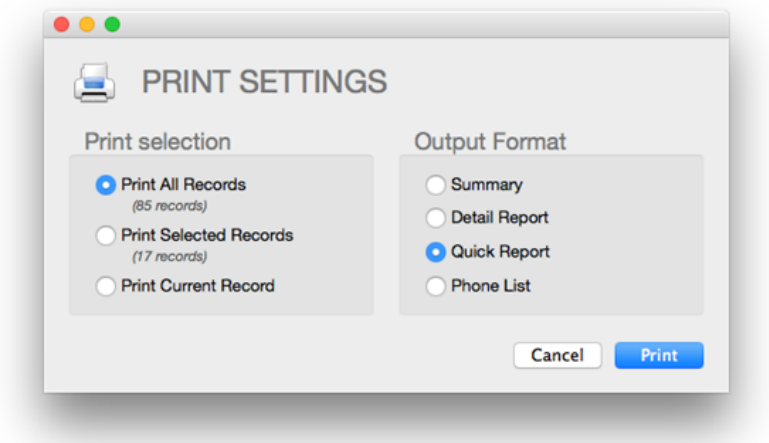
| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table owning the form or If omitted: default table or use of project form |
| form | String | ⇒ | Name of table or project form to display as dialog |
| * | Operator | ⇒ | Use the same process |

## Description

The **DIALOG** command presents the form *form* to the user. This command is often used to get information from the user through the use of variables, or to present information to the user, such as options for performing an operation.

It is common to display the form inside a modal window created with the **Open window** command.

Here is a typical example of a dialog:




Use **DIALOG** instead of **ALERT**, **CONFIRM** or **Request** when the information that must be presented or gathered is more complex than those commands can manage.

**Note:** In converted databases, it is possible to prohibit data entry in fields of dialog boxes (and thus limit data entry to variables only) using an option in the Preferences of 4D (Compatibility page). This restriction corresponds to the operation of former versions of 4D.

Unlike **ADD RECORD** or **MODIFY RECORD**, **DIALOG** does not use the current input form. You must specify the form (project form or table form) to be used in the *form* parameter. Also, the default button panel is not used if buttons are omitted. In this case, only the **Escape** (Windows) or **Esc** (Mac OS) key lets you exit the form.

The dialog is accepted if the user clicks an Accept button or presses the Enter key (numeric key pad), or if the **ACCEPT** command is executed.
Keep in mind that validation does not cause saving: if the dialog includes fields, you must explicitly call the **SAVE RECORD** command to save any data that has been modified.

The dialog is canceled if the user clicks a Cancel button or presses the cancel key (**Escape** on Windows, **Esc** on Macintosh), or if the **CANCEL** command is executed.

If you pass the optional * parameter, the form is loaded and displayed in the last open window of the current process and the command finishes it execution while leaving the active form on screen.
This form then reacts “normally” to user actions and is closed using a standard action or when 4D code related to the form (object method or form method) calls the **CANCEL** or **ACCEPT** command. If the current process terminates, the forms created in this way are automatically closed in the same way as if a **CANCEL** command had been called. This opening mode is particularly useful for displaying a floating palette with a document, without necessarily requiring another process.

**Notes:**

- You must create a window before calling the statement **DIALOG**(form;*); it is not possible to use the current dialog window in the process nor the window created by default for each process. Otherwise, the error -9909 is generated.
- When the * parameter is used, the window is closed automatically following a standard action or a call to the **CANCEL** or **ACCEPT** command. You do not have to manage the closing of the window itself.

## Example 1

The following example shows the use of **DIALOG** to specify search criteria. A custom form containing the variables *vName* and *vState* is displayed so the user can enter the search criteria.

```
Open window(10;40;370;220) ` Open a modal window
DIALOG("Search Dialog") ` Display a custom search dialog
CLOSE WINDOW ` No longer need the modal window
If(OK=1) ` If the dialog is accepted
   QUERY([Company];[Company]Name=vName;*)
   QUERY([Company];&;[Company]State=vState)
End if
```

## Example 2

The following example can be used to create a tool palette:

```
 `Display tool palette
$palette_window:=Open form window("tools";Palette form window)
DIALOG("tools";*) `Give back the control immediately
 `Display main document windowl
$document_window:=Open form window("doc";Plain form window)
DIALOG("doc")
```

## System variables and sets

After a call to **DIALOG**, if the dialog is accepted, OK is set to 1; if it is canceled, OK is set to 0.

# Modified

Modified ( aField ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ⇒ | Field to test |
| Function result | Boolean | ⇐ | True if the field has been assigned a new value, otherwise False |

## Description

**Modified** returns **True** if *field* has been programmatically assigned a value or has been edited during data entry. The **Modified** command must only be used in a form method (or a subroutine called by a form method).

Be careful, this command only returns a significant value within the same execution cycle. It is more particularly set to **False** for all the form events that correspond to the former **_o_During** execution cycle (On Clicked, On After Keystroke, etc.).

During data entry, a field is considered modified if the user has edited the field (whether or not the original value is changed) and then left it by going to another field or by clicking on a control. Note that just tabbing out of a field does not set **Modified** to True. The field must have been edited in order for **Modified** to be True.

When executing a method, a field is considered to be modified if it has been assigned a value (different or not).

**Note: Modified** always returns **True** after the execution of the **PUSH RECORD** and **POP RECORD** commands.

In all cases, use the **Old** command to detect whether the field value has actually been changed.

**Note:** Although **Modified** can be applied to any type of field, if you use it in combination with the **Old** command, be aware of the restrictions that apply to the **Old** command. For details, see the description of the **Old** command.

During data entry, it is usually easier to perform operations in object methods using the **Form event** command than to use **Modified** in form methods. Since an object method is sent an On Data Change event whenever a field is modified, the use of an object method is equivalent to using **Modified** in a form method.

**Note:** To operate properly, the **Modified** command is to be used only in a form method or in a method called by a form method.

## Example 1

The following example tests whether either the *[Orders]Quantity* field or the *[Orders]Price* field has changed. If either has been changed, then the *[Orders]Total* field is recalculated.

```
If((Modified([Orders]Quantity)|(Modified([Orders]Price))
   [Orders]Total :=[Orders]Quantity*[Orders]Price
End if
```

Note that the same thing could be accomplished by using the second line as a subroutine called by the object methods for the *[Orders]Quantity* field and the *[Orders]Price* field within the On Data Change form event.

## Example 2

You select a record for the table *[anyTable]*, then you call multiple subroutines that may modify the field *[anyTable]Important field,* but do not save the record. At the end of the main method, you can use the **Modified** command to detect whether you must save the record:

```
  ` Here the record has been selected as current record
  ` Then you perform actions using subroutines
DO SOMETHING
DO SOMETHING ELSE
DO NOT FORGET TO DO THAT
  ` ...
  ` And then you test the field to detect whether the record has to be saved
If(Modified([anyTable]Important field))
   SAVE RECORD([anyTable])
End if
```

# MODIFY RECORD

MODIFY RECORD ( {aTable}{;}{*} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to use for data entry, or Default table, if omitted |
| * | | ⇒ | Hide scroll bars |

## Description

The **MODIFY RECORD** command modifies the current record for the table *aTable* or for the default table if you omit the *aTable* parameter. **MODIFY RECORD** loads the record, if it is not already loaded for the current process, and displays the current input form. If there is no current record, then **MODIFY RECORD** does nothing. **MODIFY RECORD** does not affect the current selection.

The form is displayed in the frontmost window of the process. The window has scroll bars and a size box. Specifying the optional * parameter causes the window to be drawn without scroll bars or a size box.

To use **MODIFY RECORD**, the current record must have read-write access and should not be locked.
If the form contains buttons for moving within the selection of records, **MODIFY RECORD** lets the user click the buttons to modify records and move to other records.

The record is saved (accepted) if the user clicks an Accept button or presses the Enter key (numeric key pad), or if the **ACCEPT** command is executed.

The record is not saved (canceled) if the user clicks a Cancel button or presses the cancel key combination (Ctrl-Period on Windows, Command-Period on Macintosh), or if the **CANCEL** command is executed. Even when canceled, the record remains in memory and can be saved if **SAVE RECORD** is executed before the current record pointer is changed.

After a call to **MODIFY RECORD**, OK is set to 1 if the record is accepted, to 0 if canceled.

**Note:** Even when canceled, the record remains in memory and can be saved if **SAVE RECORD** is executed before the current record pointer is changed.

If you are using **MODIFY RECORD** and the user does not change any of the data in the record, the record is not considered to be modified, and accepting the record does not cause it to be saved again. Actions such as changing variables, checking check boxes, and selecting radio buttons do not qualify as modifications. Only changing data in a field, either through data entry or through a method, causes the record to be saved.

## Example

See example for the **ADD RECORD** command.

## System variables and sets

Accepting the record sets the OK system variable to 1; canceling it sets the OK system variable to 0. The OK system variable is set only after the record is accepted or canceled.

# Old

Old ( aField ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ⇒ | Field for which to return old value |
| Function result | Expression | ⇐ | Original field value |

## Description

The **Old** command returns the value held in *aField* before the field was programmatically assigned a value or modified in data entry.

Each time you change the current record for a table, 4D creates and maintains in memory a duplicated “image” of the new current record when it is loaded in memory. When modifying a record, you work with the actual image of the record, not this duplicated image. This image is then discarded when you change the current record again.

**Old** returns the value from the duplicated image. In other words, for an existing record, it returns the value of the field as it is stored on disk. If a record is new, **Old** returns the default empty value for *field* according to its type. For example, if *field* is an Alpha field, **Old** returns an empty string. If *field* is a numeric field, **Old** returns zero (0), and so on.

**Old** works on *aField* whether the field has been modified by a method or by the user during data entry. It can be applied to all field types.

To restore the original value of a field, assign it the value returned by **Old**.

**Note:** For technical reasons, in the case of Picture and BLOB type fields, the expression returned by **Old** cannot be used directly as a parameter for another command. It is necessary to pass the value via an intermediate variable. For example:

```
 `Do NOT write (causes a syntax error):
$size :=BLOB size(Old([theTable]theBlob)) `INCORRECT

 `Write:
$oldBLOB:=Old([theTable]theBlob)
$size :=BLOB size($oldBLOB) `CORRECT
```

# REJECT

REJECT {( aField )}

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ➡ | Field to reject |

## Description

**REJECT** has two forms. The first form has no parameters. It rejects the entire data entry and forces the user to stay in the form. The second form rejects only *aField* and forces the user to stay in the field.

**Note:** You should consider the built-in data validation tools before using this command.

The first form of **REJECT** prevents the user from accepting a record that is not complete. You can achieve the same result without using **REJECT** — you associate the Enter key with a No Action button and use the **ACCEPT** and **CANCEL** commands to accept or cancel the record, after the fields have been entered correctly. It is recommended that you use this second technique and do not use the first form of **REJECT**.

If you use the first form, you execute **REJECT** to prevent the user from accepting a record, usually because the record is not complete or has inaccurate entries. If the user tries to accept the record, executing **REJECT** prevents the record from being accepted; the record remains displayed in the form. The user must continue with data entry until the record is acceptable, or cancel the record.

The best place to put this form of **REJECT** is in the object method of an Accept button associated with the Enter key. This way, validation occurs only when the record is accepted, and the user cannot bypass the validation by pressing the Enter key.

The second form of **REJECT** is executed with the *field* parameter. In this case, the cursor stays in the field area, which forces the user to enter a correct value.
With this syntax, it is imperative that you call the **REJECT** command in the On Data Change form event.You need to put this syntax of the **REJECT** command either in the form method, or in the object method of the entry area. If you are using **REJECT** for the subform's Detail Form for a table, put it in the form method or object method for the Detail Form. This command has no effect on fields in subform areas.

You can use **HIGHLIGHT TEXT** to select the data in the field that is being rejected.

## Example 1

The following example is for a bank transaction record. It shows the first form of **REJECT** being used in an Accept button object method. The Enter key is set as an equivalent for the button. This means that even if the user presses the Enter key to accept the record, the button's object method will be executed. If the transaction is a check, then there must be a check number. If there is no check number, the validation is rejected:

```
Case of
   :(([Operation]Transaction="Check") & ([Operation]Check Number=""))  ` If it is a check with no number...
      ALERT("Please fill in the check number.")  ` Alert the user
      REJECT  ` Reject the entry
      GOTO OBJECT([Operation]Check Number)  ` Go to the check number field
End case
```

## Example 2

The following example is part of an object method for an *[Employees]Salary* field. The object method tests the *Employees]Salary* field and rejects the field if it is less than $10,000. You could perform the same operation by specifying a minimum value for the field in the form editor:

```
Case of
   :(Form event=On Data Change)
      If([Employees]Salary<10000)
         ALERT("Salary must be greater than $10,000")
         REJECT([Employees]Salary)
```

```
        End if
    End case
```

# _o_ADD SUBRECORD

_o_ADD SUBRECORD ( subtable ; form {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ➡ | Subtable to use for data entry |
| form | String | ➡ | Form to use for data entry |
| * | | ➡ | Hide scroll bars |

## Compatibility Note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_MODIFY SUBRECORD

_o_MODIFY SUBRECORD ( subtable ; form {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ➡ | Subtable to use for data entry |
| form | String | ➡ | Form to use for data entry |
| * | | ➡ | Hide scroll bars |

## Compatibility Note

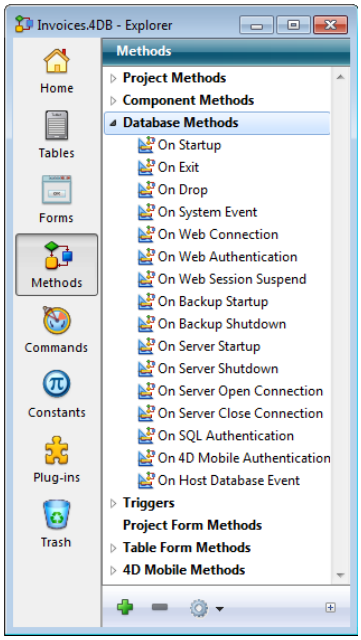
Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# Database Methods

- Database Methods
- On 4D Mobile Authentication database method
- On Backup Shutdown database method
- On Backup Startup database method
- On Drop database method
- On Exit database method
- On Host Database Event database method
- On Server Close Connection database method
- On Server Open Connection database method
- On Server Shutdown database method
- On Server Startup database method
- On SQL Authentication database method
- On Startup database method
- On System Event database method
- On Web Authentication database method
- On Web Close Process database method
- On Web Connection database method

# Database Methods

Database methods are methods that are automatically executed by 4D when a general session event occurs.




To create or open and edit a database method:

1. Open the **Explorer** window.
2. Select the **Methods** page.
3. Expand the **Database Methods** theme.
4. Double click on the method.

or:

1. Select the method.
2. Press Enter or Return.

You edit a database method in the same way as any other method.

You cannot call a database method from another method. Database methods are automatically invoked by 4D at certain points in a working session. The following table summarizes the execution of database methods:

| Database Method | 4D local | 4D Server | 4D remote |
|---|---|---|---|
| On Startup | Yes, Once | No | Yes, Once |
| On Exit | Yes, Once | No | Yes, Once |
| On Drop | Yes, Multiple | No | Yes, Multiple |
| On System Event | Yes, Multiple | Yes, Multiple | Yes, Multiple |
| On Web Connection | Yes, Multiple | Yes, Multiple | Yes, Multiple |
| On Web Authentication | Yes, Multiple | Yes, Multiple | Yes, Multiple |
| On Web Session Suspend | Yes, Multiple | Yes, Multiple | Yes, Multiple |
| On Backup Startup | Yes, Multiple | Yes, Multiple | Yes, Multiple |
| On Backup Shutdown | Yes, Multiple | Yes, Multiple | Yes, Multiple |
| On Server Startup | No | Yes, Once | No |
| On Server Shutdown | No | Yes, Once | No |
| On Server Open Connection | No | Yes, Multiple | No |
| On Server Close Connection | No | Yes, Multiple | No |
| On SQL Authentication | Yes, Multiple | Yes, Multiple | Yes, Multiple |
| On 4D Mobile Authentication | Yes, Multiple | Yes, Multiple | Non |
| On Host Database Event | Yes, Multiple | Yes, Multiple | Yes, Multiple |

# On 4D Mobile Authentication database method

$1, $2, $3 -> On 4D Mobile Authentication database method -> $0

| Parameter | Type | | Description |
|---|---|---|---|
| $1 | Text | ⇦ | User name |
| $2 | Text | ⇦ | Password |
| $3 | Boolean | ⇦ | True = Digest mode, False = Basic mode |
| $0 | Boolean | ⇨ | True = request accepted, False = request rejected |

## Description

The **On 4D Mobile Authentication database method** provides you with a custom way of controlling the opening of 4D Mobile sessions (via REST) on 4D. This database method is mainly intended for filtering connections when setting up a connection between a Wakanda Server and 4D.

When the request to open a 4D Mobile session comes from Wakanda Server by means of the ***mergeOutsideCatalog()*** method (general case), the connection identifiers are provided in the header of the request. The **On 4D Mobile Authentication database method** is called so that you can evaluate these identifiers. You can use the list of users for the 4D database or you can use your own table of identifiers.

**Important:** When **On 4D Mobile Authentication database method** is defined (i.e. when it contains code), 4D fully delegates control of 4D Mobile requests to it: any setting made using the "Read/Write" menu on the Web/4D Mobile page of the Database Settings is ignored (see the *Design Reference* manual).

The database method receives two parameters (*$1* and *$2*) of the Text type and a Boolean (*$3*), passed by 4D, and returns a Boolean, *$0*. You must declare these parameters as follows:

```
 //On 4D Mobile Authentication database method
C_TEXT($1;$2)
C_BOOLEAN($0;$3)
... // Code for the method
```

*$1* contains the user name and *$2* the password used for the connection.

The password (*$2*) can be received either in clear or hashed form, depending mode used by the request. This mode is indicated by the *$3* parameter to enable you to perform the appropriate processing:

- If the password is sent in clear (Basic mode), *$3* returns **False**.
- If it is sent in hashed form (Digest mode), *$3* returns **True**.

When a 4D Mobile connection request comes from Wakanda Server, the password is always sent in hashed form.

You must check the identifiers of the 4D Mobile connection in the database method. Usually, you check the name and password using a custom user table. If the identifiers are valid, pass **True** in *$0*. The request is then accepted; 4D executes it and returns the result in JSON.
Otherwise, pass **False** in *$0*; in this case, the connection is rejected and the server returns an authentication error to the sender of the request.

If the user is referenced in the list of 4D users of the database, you can check the password directly by means of the following statement:

```
$0:=Validate password($1;$2;$3)
```

The **Validate password** command has been extended to accept a user name as first parameter as well as an optional parameter indicating whether the password is expressed in hashed form.

If you want to use your own list of users external to the 4D database list, you can save their passwords in hashed form using the same algorithm as that used by Wakanda Server when sending the connection request to the **On 4D Mobile Authentication database method** in *$2*. To hash a password using this method, you can write:

```
$HashedPasswd :=Generate digest($ClearPasswd ;4D digest)
```

The **Generate digest** command accepts 4D digest as a hashing algorithm, corresponding to the method used by 4D for its internal management of passwords.

## Example 1

This example only accepts the "admin" user with the password "123" that does not match a 4D user:

```
 //On 4D Mobile Authentication database method
C_TEXT($1;$2)
C_BOOLEAN($0;$3)
 //$1: user
 //$2: password
 //$3: digest mode
If($1="admin")
   If($3)
      $0:=($2=Generate digest("123";4D digest))
   Else
      $0:=($2="123")
   End if
Else
   $0:=False
End if
```

## Example 2

This example of the **On 4D Mobile Authentication database method** checks that the connection request comes from one of the two authorized Wakanda servers, saved in the users of the 4D database:

```
C_TEXT($1;$2)
C_BOOLEAN($0)
ON ERR CALL("4DMOBILE_error")
If($1="WAK1")|($1="WAK2")
   $0:=Validate password($1;$2;$3)
Else
   $0:=False
End case
```

# On Backup Shutdown database method

$1 -> On Backup Shutdown database method

| Parameter | Type | | Description |
|---|---|---|---|
| $1 | Longint | ⇦ | 0 = backup executed correctly; other value = error, interrupted by user or code returned by On Backup Startup |

---

The **On Backup Shutdown database method** is called every time a database backup ends. The reasons for the stoppage of a backup can be the end of the copy, user interruption or an error.
This concerns all 4D environments: 4D (all modes), 4D Server as well as 4D applications compiled and merged with 4D Volume Desktop.

The **On Backup Shutdown database method** allows verifying that the backup was executed correctly. It receives, in the *$1* parameter, a value representing the status of the backup once completed:

- If the backup was executed correctly, *$1* equals 0.
- If the backup was interrupted by the user or following an error, *$1* is different from 0.
  - If the backup was stopped by the **On Backup Startup Database Method** (*$0* # 0), *$1* gets the value actually returned in the *$0* parameter. This allows you to implement a customized error management system.
  - If the backup was stopped due to an error, the error code is returned in *$1*.

  In any case, you can get information about the error using the **GET BACKUP INFORMATION** command.

**Note**: You must declare the *$1* parameter (longint) in the database method:

```
C_LONGINT($1)
```

It is important to note that in the case of an error during backup (disk full, support unavailable, etc.), the information related to the error is only displayed in the 4D Server monitor or in the MSC, and copied into the backup log. No alert dialog box appears and the *error* variable is not modified. If you want to be able to notify the administrator that an error has occurred, particularly in the context of an application running in client/server mode, you will need to use the **On Backup Shutdown database method**.

# On Backup Startup database method

On Backup Startup database method -> $0

| Parameter | Type | | Description |
|---|---|---|---|
| $0 | Longint | ⮌ | 0 = backup can be launched; value other than 0 = backup not authorized |

---

The **On Backup Startup database method** is called every time a database backup is about to start (manual backup, scheduled automatic backup, or using the **BACKUP** command).
This concerns all 4D environments: 4D (all modes), 4D Server and databases merged with 4D Volume Desktop.

The **On Backup Startup database method** allows verifying that the backup started. In this method, you should return a value that authorizes or refuses the backup in the $0 parameter:

- If $0 = 0, the backup can be launched.
- If $0 # 0, the backup is not authorized. The operation is cancelled and an error is returned. You can get the error using the **GET BACKUP INFORMATION** command.

You can use this database method to verify backup execution conditions (user, date of the last backup, etc.).

**Note:** You must declare the *$0* parameter (longint) in the database method:

```
C_LONGINT($0)
```

# On Drop database method

On Drop database method
Does not require any parameters

---

The **On Drop database method** is available in local or remote 4D applications.

This database method is automatically executed in the case of objects being dropped in the 4D application outside of any form or windows, i.e.:

- In an empty area of the MDI window (Windows),
- On the 4D icon in the Dock (Mac OS) or on the system desktop.

Under Mac OS, you need to hold down the **Option**+**Command** keys during the drop in order for the database method to be called.

When a drop occurs on the 4D application icon on the desktop, the **On Drop database method** is only called when the application is already launched, except in the case of applications merged with 4D Desktop. In this case, the database method is called even when the application is not launched. This means that it is possible to define custom document signatures.

## Example

This example can be used to open a 4D Write document that is dropped outside of any form:

```
  `On Drop database method
droppedFile:=Get file from pasteboard(1)
If(Position(".4W7";droppedFile)=Length(droppedFile)-3)
   externalArea:=Open external window(100;100;500;500;0;droppedFile;"_4D Write")
   WR OPEN DOCUMENT(externalArea;droppedFile)
End if
```

# On Exit database method

On Exit database method
Does not require any parameters

---

The **On Exit database method** is called once when you quit a database.

This method is used in the following 4D environments:

- 4D in local mode
- 4D in remote mode
- 4D application compiled and merged with 4D VolumeDesktop

**Note**: The **On Exit database method** is NOT invoked by 4D Server.

The **On Exit database method** is automatically invoked by 4D; unlike project methods, you cannot call this database method yourself by programming. You can however execute it from the Method editor. You can also use subroutines.

A database can be exited if any of the following occur:

- The user selects the menu command **Quit** from the Design Environment **File** menu or from the Application environment (Quit standard action)
- A call to the **QUIT 4D** command is issued
- A 4D Plug-in issues a call to the **QUIT 4D** entry point

No matter how the exit from the database was initiated, 4D performs the following actions:

- If there is no **On Exit database method**, 4D aborts each running process one by one, without distinction. If the user is performing data entry, the records will be cancelled and not saved.
- If there is an **On Exit database method**, 4D starts executing this method within a newly created local process. You can therefore use this database method to inform other processes, via interprocess communication, that they must close (data entry) or stop executing. Note that 4D will eventually quit—the **On Exit database method** can perform all the cleanup or closing operations you want, but it cannot refuse the quit, and will at some point end.

The **On Exit database method** is the perfect place to:

- Stop processes automatically started when the database was opened
- Save (locally, on disk) Preferences or Settings to be reused at the beginning of the next session in the **On Startup database method**
- Perform any other actions that you want to be done automatically each time a database is exited

**Note:** Don't forget that the **On Exit database method** is a local/client process, so it cannot access the data file. Thus, if the **On Exit database method** performs a query or a sort, a 4D Client that is about to quit will "freeze" and actually will not quit. If you need to access data when a client quits the application, create a new global process from within the **On Exit database method**, which will be able to access the data file. In this case, be sure that the new process will terminate correctly before the end of the **On Exit database method** execution (by using interprocess variables, by example).

## Example

The following example covers all the methods used in a database that tracks the significant events that occur during a working session and writes a description in a text document called "Journal."

- The **On Startup database method** initializes the interprocess variable *◊vbQuit4D*, which tells all the use processes whether or not the database is being exited. It also creates the journal file, if it does not already exist.

```
` On Startup Database Method
C_TEXT(◊vtIPMessage)
C_BOOLEAN(◊vbQuit4D)
◊vbQuit4D:=False
```

```
If(Test path name("Journal")#Is a document)
   $vhDocRef:=Create document("Journal")
   If(OK=1)
      CLOSE DOCUMENT($vhDocRef)
   End if
End if
WRITE JOURNAL("Opening Session")
```

- The project method ***WRITE JOURNAL***, used as subroutine by the other methods, writes the information it receives, in the journal file:

```
  ` WRITE JOURNAL Project Method
  ` WRITE JOURNAL ( Text )
  ` WRITE JOURNAL ( Event description )
C_TEXT($1)
C_TIME($vhDocRef)

While(Semaphore("$Journal"))
   DELAY PROCESS(Current process;1)
End while
$vhDocRef:=Append document("Journal")
If(OK=1)
   PROCESS PROPERTIES(Current process;$vsProcessName;$vlState;$vlElapsedTime;$vbVisible)
   SEND PACKET($vhDocRef;String(Current date)+Char(9)+String(Current time)+Char(9)
   +String(Current process)+Char(9)+$vsProcessName+Char(9)+$1+Char(13))
   CLOSE DOCUMENT($vhDocRef)
End if
CLEAR SEMAPHORE("$Journal")
```

Note that the document is open and closed each time. Also note the use of a semaphore as “access protection” to the document—we do not want two processes trying to access the journal file at the same time.

- The ***M_ADD_RECORDS*** project method is executed when a menu item **Add Record** is chosen in the Application environment:

```
  ` M_ADD_RECORDS Project Method
SET MENU BAR(1)
Repeat
   ADD RECORD([Table1];*)
   If(OK=1)
      WRITE JOURNAL("Adding record #"+String(Record number([Table1]))+" in Table1")
   End if
Until((OK=0)|◊vbQuit4D)
```

This method loops until the user cancels the last data entry or exits the database.

- The input form for *[Table 1]* includes the treatment of the On Outside Call events. So, even if a process is in data entry, it can be exited smoothly, with the user either saving (or not saving) the current data entry:

```
  ` [Table1];"Input" Form Method
Case of
   :(Form event=On Outside Call)
      If(◊vtIPMessage="QUIT")
         CONFIRM("Do you want to save the changes made to this record?")
         If(OK=1)
            ACCEPT
         Else
            CANCEL
         End if
      End if
End case
```

- The *M_QUIT* project method is executed when **Quit** is chosen from the **File** menu in the Application environment:

```
  ` M_QUIT Project Method
$vlProcessID:=New process("DO_QUIT";32*1024;"$DO_QUIT")
```

The method uses a trick. When **QUIT 4D** is called, the command has an immediate effect. Therefore, the process from which the call is issued is in "stop mode" until the database is actually exited. Since this process can be one of the processes in which data entry occurs, the call to **QUIT 4D** is made in a local process that is started only for this purpose. Here is the *DO_QUIT* method:

```
  ` DO_QUIT Project Method
CONFIRM("Are you sure you want to quit?")
If(OK=1)
   WRITE JOURNAL("Quitting Database")
   QUIT 4D
  ` QUIT 4D has an immediate effect, any line of code below will never be executed
  ` ...
End if
```

- Finally, here is the **On Exit database method** which tells all open user processes "It's time to get out of here!" It sets *◊vbQuit4D* to *True* and sends interprocess messages to the user processes that are performing data entry:

```
  ` On Exit Database Method
◊vbQuit4D:=True
Repeat
   $vbDone:=True
   For($vlProcess;1;Count tasks)
      PROCESS PROPERTIES($vlProcess;$vsProcessName;$vlState;$vlElapsedTime;$vbVisible)
      If((((($vsProcessName="ML_@")|($vsProcessName="M_@")))&($vlState>=0))
         $vbDone:=False
         ◊vtIPMessage:="QUIT"
         BRING TO FRONT($vlProcess)
         CALL PROCESS($vlProcess)
         $vhStart:=Current time
         Repeat
            DELAY PROCESS(Current process;60)
         Until((Process state($vlProcess)<0)|((Current time-$vhStart)>=?00:01:00?))
      End if
   End for
Until($vbDone)
WRITE JOURNAL("Closing session")
```

**Note**: Processes that have names beginning with *"ML_..."* or *"M_..."* are started by menu commands for which the **Start a New Process** property has been selected. In this example, these are the processes started when the menu command **Add record** was chosen.

The test *(Current time-$vhStart)>=?00:01:00?* allows the database method to get out of the "waiting the other process" Repeat loop if the other process does not act immediately.
The following is a typical example of the Journal file produced by the database:

| | | | | |
|---|---|---|---|---|
| 2/6/03 | 15:47:25 | 1 | Main process | Opening Session |
| 2/6/03 | 15:55:43 | 5 | ML_1 | Adding record #23 in Table1 |
| 2/6/03 | 15:55:46 | 5 | ML_1 | Adding record #24 in Table1 |
| 2/6/03 | 15:55:54 | 6 | $DO_QUIT | Quitting Database |
| 2/6/03 | 15:55:58 | 7 | $xx | Closing session |

**Note**: The name *$xx* is the name of the local process started by 4D in order to execute the **On Exit database method**.

# On Host Database Event database method

$1 -> On Host Database Event database method

| Parameter | Type | | Description |
|---|---|---|---|
| $1 | Longint | ⇦ | Event code |

## Description

The **On Host Database Event database method** allows 4D components to execute code when the host database is opened and closed.

**Note:** For security reasons, in order to be able to call this database method, you must explicitly allow its execution in the host database. For more information about this point, refer to the *Design Reference* manual.

The **On Host Database Event database method** is executed automatically only in databases used as components of host databases (when it is authorized in the Settings of the host database). It is called when events related to the opening and closing of the host database occur.

To process an event, you must test the value of the *$1* parameter inside the method, and compare it with one of the following constants, available in the "**Database Events**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| On after host database exit | Longint | 4 | The **On Exit database method** of the host database has just finished running |
| On after host database startup | Longint | 2 | The **On Startup database method** of the host database just finished running |
| On before host database exit | Longint | 3 | The host database is closing. The **On Exit database method** of the host database has not yet been called.<br>The **On Exit database method** of the host database is not called while the **On Host Database Event database method** of the component is running |
| On before host database startup | Longint | 1 | The host database has just been started. The **On Startup database method** method of the host database has not yet been called.<br>The On Startup database method of the host database is not called while the **On Host Database Event database method** of the component is running |

This allows 4D components to load and save preferences or user states related to the operation of the host database.

## Example

Example of typical structure of an On Host Database Event database method:

```
 // On Host Database Event database method
C_LONGINT($1)
Case of
   :($1=On before host database startup)
 // put code here that you want to execute before the "On Startup" database method
 // of the host database
   :($1=On after host database startup)
 // put code here that you want to execute after the "On Startup"
 // database method of the host database
   :($1=On before host database exit)
 // put code here that you want to execute before the "On Exit"
 // database method of the host database
   :($1=On after host database exit)
 // put code here that you want to execute after the "On Exit"
 // database method of the host database
End case
```

# On Server Close Connection database method

$1, $2, $3 -> On Server Close Connection database method

| Parameter | Type | | Description |
|---|---|---|---|
| $1 | Longint | ⇦ | User ID number used internally by 4D Server to identify users |
| $2 | Longint | ⇦ | Connection ID number used internally by 4D Server to identify a connection |
| $3 | Longint | ⇦ | Obsolete: Always returns 0 but must be declared |

## Description

The **On Server Close Connection database method** is called once on the Server machine each time a 4D Client process ends.

As for the **On Server Open Connection database method**, 4D Server passes three Long Integer parameters to the **On Server Close Connection database method**. On the other hand, no result is expected by 4D Server.

The method must therefore be explicitly declared with three Long Integer parameters:

```
C_LONGINT($1;$2;$3)
```

This table details the information provided by the three parameters passed to the database method:

| Parameter | Description |
|---|---|
| $1 | User ID number used internally by 4D Server to identify users |
| $2 | Connection ID number used internally by 4D Server to identify a connection |
| $3 | Obsolete: Always returns 0 but must be declared |

The **On Server Close Connection database method** is the exact counterpoint to the **On Server Open Connection database method**. For more information and a description of the **4D Client processes**, see the description of this database method.

## Example

See the first example for **On Server Open Connection database method**.

# On Server Open Connection database method

$1, $2, $3 -> On Server Open Connection database method -> $0

| Parameter | Type | | Description |
|---|---|---|---|
| $1 | Longint | ⇦ | User ID number used internally by 4D Server to identify users |
| $2 | Longint | ⇦ | Connection ID number used internally by 4D Server to identify a connection |
| $3 | Longint | ⇦ | Obsolete: Always returns 0 (but must be declared) |
| $0 | Longint | ⇨ | 0 or omitted = connection accepted; other value = connection refused |

## When is the On Server Open Connection Database Method Called?

The **On Server Open Connection database method** is called once on the Server machine each time a connection process is started by a 4D remote workstation. The **On Server Open Connection database method** is NOT invoked by any 4D environment other than 4D Server.

The **On Server Open Connection database method** is called each time:

- a remote 4D connects (because the Application process starts)
- a remote 4D opens the Design environment (because the Design process starts)
- a remote 4D starts a global process (whose name does not begin with "$") which requires the creation of a cooperative process on the server **(*)**. This process can be created using the **New process** command, a menu command or using the Execute Method dialog box.

In each case with a remote 4D, several processes are started—One on the client machine and one or two others (as needed) on the server machine. On the client machine, the process executes code and send requests to 4D Server. On the server machine, the **4D Client Process** (preemptive process) maintains the database environment for the client process (i.e., current selections and locking of records for user processes) and replies to requests sent by the process running on the client machine. The **4D Client Database process** (cooperative process) is in charge of monitoring the corresponding 4D Client process.

**(*)** Beginning with 4D v13, for optimization purposes, the server processes (a preemptive process for access to the database engine and a cooperative process for access to the language) are only created when necessary when executing client-side code. For example, here are the details of a 4D code sequence running in a new client process:

```
 // global process begins without a new process on the server, like a local process.
CREATE RECORD([Table_1])
[Table_1]field1_1:="Hello world"
SAVE RECORD([Table_1]) // creation here of preemptive process on server
$serverTime:=Current time(*) // creation here of cooperative process on server
 // call to On Server Open Connection
```

**Important**: Web connections and SQL connections do not invoke the **On Server Open Connection database method**. When a Web browser connects to 4D Server, the **On Web Authentication Database Method** (if any) and/or the **On Web Connection database method** are invoked. When 4D Server receives an SQL query, the **On SQL Authentication database method** (if one exists) is called. For more information, see the description of this database method in the 4D Language Reference manual.

**Important**: When a Stored Procedure is started, the **On Server Open Connection database method** is NOT invoked. **Stored Procedures** are server processes, not 4D Client processes. They execute code on the Server machine, but do not reply to requests exchanged by a 4D client (or other clients) and 4D Server.

## How is the On Server Open Connection Database Method Called?

The **On Server Open Connection database method** is executed on the 4D Server machine within the 4D Client process that provoked the call to the method.

For example, if a remote 4D connects to a 4D Server interpreted database, the user process, the Design process and the client registration process (by default) for that client are started. The **On Server Open Connection database method** is therefore executed three times in a row—the first time within the Application process, the second time within the client registration process, and the third time within the Design process. If the three process are respectively the sixth, seventh

and eighth process to be started on the Server machine, and if you call **Current process** from within the **On Server Open Connection database method**, the first time **Current process** returns 6, the second time 7 and the third time 8.

Note that **On Server Open Connection database method** executes on the Server machine. It executes within the 4D Client process running on the Server machine, independent of the process running on the client side. In addition, at the moment when the method is invoked, the 4D Client process has not yet been named (**PROCESS PROPERTIES** will not at this point return the name of the 4D Client process).

The **On Server Open Connection database method** has no access to the process variable table of the process running on the Client side. This table resides on the Client machine, not on the Server machine.

When the **On Server Open Connection database method** accesses a process variable, it works with a private and dynamically created process variable table for the 4D Client process.

4D Server passes three Long Integer parameters to the **On Server Open Connection database method** and expects a Long Integer result. The method must therefore be explicitly declared with three Long Integer parameters as well as a Long Integer function result:

```
C_LONGINT($0;$1;$2;$3)
```

If you do not return a value in *$0*, thereby leaving the variable undefined or initialized to zero, 4D Server assumes that the database method accepts the connection. If you do not accept the connection, you return a non-null value in *$0*.

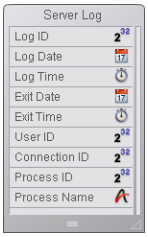
This table details the information provided by the three parameters passed to the database method:

| Parameter | Description |
|---|---|
| $1 | User ID number used internally by 4D Server to identify users |
| $2 | Connection ID number used internally by 4D Server to identify a connection |
| $3 | Obsolete: Always returns 0 but must be declared |

These ID numbers are not directly usable as sources of information to be passed as, for example, parameters to a 4D command. However, they provide a way to uniquely identify a 4D Client process between the **On Server Open Connection database method** and the **On Server Close Connection database method**. At any moment of a 4D Server session, the combination of these values is unique. By storing this information in an interprocess array or a table, the two database methods can exchange information. In the example at the end of this section, the two database methods use this information to store the date and time of the beginning and end of a connection in the same record of a table.

## Example 1

The following example shows how to maintain a log of the connections to the database using the **On Server Open Connection database method** and the **On Server Close Connection Database Method**. The *[Server Log]* table (shown below) is used to keep track of the connection processes:

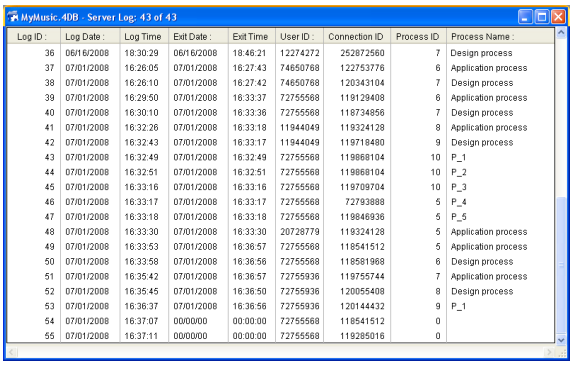



The information stored in this table is managed by the **On Server Open Connection database method** and the **On Server Close Connection Database Method** listed here:

```
  ` On Server Open Connection Database Method
C_LONGINT($0;$1;$2;$3)
  ` Create a [Server Log] record
CREATE RECORD([Server Log])
[Server Log]Log ID:=Sequence number([Server Log])
  ` Save the Log Date and Time
[Server Log]Log Date:=Current date
[Server Log]Log Time:=Current time
  ` Save the connection information
[Server Log]User ID:=$1
[Server Log]Connection ID:=$2
SAVE RECORD([Server Log])
  ` Returns no error so that the connection can continue
$0:=0

  ` On Server Close Connection Database Method
```

```
C_LONGINT($1;$2;$3)
  ` Retrieve the [Server Log] record
QUERY([Server Log];[Server Log]User ID=$1;*)
QUERY([Server Log]; & ;[Server Log]Connection ID=$2;*)
QUERY([Server Log]; & ;[Server Log]Process ID=0)
  ` Save the Exit date and time
[Server Log]Exit Date:=Current date
[Server Log]Exit Time:=Current time
  ` Save the process information
[Server Log]Process ID:=Current process
PROCESS PROPERTIES([Server Log]Process ID;$vsProcName;$vlProcState;$vlProcTime)
[Server Log]Process Name:=$vsProcName
SAVE RECORD([Server Log])
```

Here are some entries in the [Server Log] showing several remote connections:



MyMusic.4DB - Server Log: 43 of 43

| Log ID : | Log Date : | Log Time | Exit Date : | Exit Time | User ID : | Connection ID | Process ID | Process Name : |
|---|---|---|---|---|---|---|---|---|
| 36 | 06/16/2008 | 18:30:29 | 06/16/2008 | 18:46:21 | 12274272 | 252872560 | 7 | Design process |
| 37 | 07/01/2008 | 16:26:05 | 07/01/2008 | 16:27:43 | 74650768 | 122753776 | 6 | Application process |
| 38 | 07/01/2008 | 16:26:10 | 07/01/2008 | 16:27:42 | 74650768 | 120343104 | 7 | Design process |
| 39 | 07/01/2008 | 16:29:50 | 07/01/2008 | 16:33:37 | 72755568 | 119129408 | 6 | Application process |
| 40 | 07/01/2008 | 16:30:10 | 07/01/2008 | 16:33:36 | 72755568 | 118734856 | 7 | Design process |
| 41 | 07/01/2008 | 16:32:26 | 07/01/2008 | 16:33:18 | 11944049 | 119324128 | 8 | Application process |
| 42 | 07/01/2008 | 16:32:43 | 07/01/2008 | 16:33:17 | 11944049 | 119718480 | 9 | Design process |
| 43 | 07/01/2008 | 16:32:49 | 07/01/2008 | 16:32:49 | 72755568 | 119868104 | 10 | P_1 |
| 44 | 07/01/2008 | 16:32:51 | 07/01/2008 | 16:32:51 | 72755568 | 119868104 | 10 | P_2 |
| 45 | 07/01/2008 | 16:33:16 | 07/01/2008 | 16:33:16 | 72755568 | 119709704 | 10 | P_3 |
| 46 | 07/01/2008 | 16:33:17 | 07/01/2008 | 16:33:17 | 72755568 | 72793888 | 5 | P_4 |
| 47 | 07/01/2008 | 16:33:18 | 07/01/2008 | 16:33:18 | 72755568 | 119846936 | 5 | P_5 |
| 48 | 07/01/2008 | 16:33:30 | 07/01/2008 | 16:33:30 | 20728779 | 119324128 | 5 | Application process |
| 49 | 07/01/2008 | 16:33:53 | 07/01/2008 | 16:36:57 | 72755568 | 118541512 | 5 | Application process |
| 50 | 07/01/2008 | 16:33:58 | 07/01/2008 | 16:36:56 | 72755568 | 118581968 | 6 | Design process |
| 51 | 07/01/2008 | 16:35:42 | 07/01/2008 | 16:36:57 | 72755936 | 119755744 | 7 | Application process |
| 52 | 07/01/2008 | 16:35:45 | 07/01/2008 | 16:36:50 | 72755936 | 120055408 | 8 | Design process |
| 53 | 07/01/2008 | 16:36:37 | 07/01/2008 | 16:36:56 | 72755936 | 120144432 | 9 | P_1 |
| 54 | 07/01/2008 | 16:37:07 | 00/00/00 | 00:00:00 | 72755568 | 118541512 | 0 | |
| 55 | 07/01/2008 | 16:37:11 | 00/00/00 | 00:00:00 | 72755568 | 119285016 | 0 | |

## Example 2

The following example prevents any new connection from 2 to 4 A.M.

```
  ` On Server Open Connection Database Method
C_LONGINT($0;$1;$2;$3)

If((?02:00:00?<=Current time)&(Current time<?04:00:00?))
   $0:=22000
Else
   $0:=0
End if
```

# On Server Shutdown database method

On Server Shutdown database method
Does not require any parameters

---

The **On Server Shutdown database method** is called once on the server machine when the current database is closed on 4D Server. The **On Server Shutdown database method** is NOT invoked by any 4D environment other than 4D Server.

To close the current database on the server, you can select the **Close Database...** menu command on the server. You can also choose the **Quit** command or call the **QUIT 4D** command within a stored procedure executed on the server.

When the exit from the database is initiated, 4D performs the following actions:

- If there is no **On Server Shutdown database method**, 4D Server aborts each running process one by one, without distinction.
- If there is an **On Server Shutdown database method**, 4D Server starts executing this method within a newly created local process. You can therefore use this database method to inform other processes, via interprocess communication, that they must stop executing. Note that 4D Server will eventually quit — the **On Server Shutdown database method** can perform all the cleanup or closing operations you want, but it cannot refuse the quit, and will at some point end.

The **On Server Shutdown database method** is the perfect place to:

- Stop store procedures automatically started when the database was opened.
- Save (locally, on disk) Preferences or Settings to be reused at the beginning of the next session in the **On Server Startup Database Method**.
- Perform any other actions that you want to be done automatically each time a database is exited.

**Warning:** If you use the **On Server Shutdown database method** to close stored procedures, keep in mind that the server quits once the **On Server Shutdown database method** (and not the stored procedures) is executed. If some stored procedures are still running at this point, they will be killed.
Consequently, if you want to make sure that the stored procedures are fully executed before being killed by the server, the **On Server Shutdown database method** should indicate to the stored procedures that they must end their execution (for example, using an interprocess variable) and should allow them to close (through a x seconds loop or another interprocess variable).

If you want code to be executed automatically on a client machine when a remote 4D stops connecting to the server, use the **On Exit database method**.

# On Server Startup database method

On Server Startup database method
Does not require any parameters

---

The **On Server Startup database method** is called once on the server machine when you open a database with 4D Server.The **On Server Startup database method** is NOT invoked by any 4D environment other than 4D Server.

The **On Server Startup database method** is the perfect place to:

- Initialize interprocess variables that you will use during the whole 4D Server session.
- Start **Stored Procedures** automatically when a database is opened.
- Load Preferences or Settings saved during the previous 4D Server session.
- Prevent the opening of the database if a condition is not met (i.e., missing system resources) by explicitly calling **QUIT 4D**.
- Perform any other actions that you want performed automatically each time a database is opened.

To automatically execute code on a client machine when a remote 4D connects to the server, use the **On Startup database method**.

**Note:** The **On Server Startup database method** is executed automatically, which means that no remote 4D can connect until the method has finished executing.

# On SQL Authentication database method

$1, $2, $3 -> On SQL Authentication database method -> $0

| Parameter | Type | | Description |
|---|---|---|---|
| $1 | Text | ⇦ | User name |
| $2 | Text | ⇦ | Password |
| $3 | Text | ⇦ | (Optional) IP address of client at origin of the request |
| $0 | Boolean | ⇨ | True = request accepted, False = request refused |

---

The **On SQL Authentication database method** can be used to filter requests sent to the integrated SQL server of 4D. This filtering can be based on the name and password as well as the (optional) IP address of the user. The developer can use their own table of users or that of the 4D users to evaluate the connection identifiers. Once the connection is authenticated, the **CHANGE CURRENT USER** command must be called in order to control access of requests within the 4D database.

When it exists, the **On SQL Authentication database method** is called automatically by 4D or 4D Server on each external connection to the SQL server. The internal system for managing 4D users is therefore not activated. The connection is only accepted if the database method returns **True** in $0 and if the **CHANGE CURRENT USER** command has been executed successfully. If one of these conditions is not met, the request is refused.

**Note**: The statement **SQL LOGIN**(**SQL_INTERNAL**;$user;$password) does not call the **On SQL Authentication database method** since it is an internal connection in this case.

The database method receives up to three parameters of the Text type, passed by 4D ($1, $2 and $3), and returns a Boolean, $0. Here is a description of these parameters:

| Parameters | Type | Description |
|---|---|---|
| $1 | Text | User name |
| $2 | Text | Password |
| $3 | Text | (optional) IP address of client at origin of the request(*) |
| $0 | Boolean | True = request accepted, False = request refused |

(*) 4D returns IPv4 addresses in a hybrid IPv6/IPv4 format written with a 96-bit prefix, for example ::ffff:192.168.2.34 for the IPv4 address 192.168.2.34. For more information, refer to the **Support of IPv6** section.

You must declare these parameters as follows:

```
  ` On Web Authentication database method

C_TEXT($1;$2;$3)
C_BOOLEAN($0)
  ` Code for method
```

The password ($2) is received as standard text.

You must check the identifiers of the SQL connection in the **On SQL Authentication database method**. For example, you can check the name and password using a custom table of users. If the identifiers are valid, pass **True** in $0 to accept the connection. Otherwise, pass **False** in $0; in this case, the connection is refused.

**Note:** If the **On SQL Authentication database method** does not exist, the connection is evaluated using the integrated user management system of 4D (if it is activated, in other words, if a password has been assigned to the Designer). If this system is not activated, users are connected with Designer access rights (free access).

If you have passed **True** in $0, you must then successfully call the **CHANGE CURRENT USER** command in the **On SQL Authentication database method** in order for the request to be accepted and for 4D to open an SQL session for the user.

The use of the **CHANGE CURRENT USER** command can be used to implement a virtual authentication system which has the double advantage of allowing the control of connection actions and of hiding the connection identifiers from the outside in the 4D SQL session.

This example of the **On SQL Authentication database method** checks whether the connection request comes from the internal network, validates the identifiers and then assigns access rights to the "sql_user" user for the SQL session.

```
C_TEXT($1;$2;$3)
C_BOOLEAN($0)
```

```
   `$1: user
   `$2: password
   `{$3: IP address of client}
ON ERR CALL("SQL_error")
If(checkInternalIP($3))
   `The checkInternalIP method checks whether the IP address is internal
   If($1="victor") & ($2="hugo")
      CHANGE CURRENT USER("sql_user";"")
      If(OK=1)
         $0:=True
      Else
         $0:=False
      End if
   Else
      $0:=False
   End if
Else
   $0:=False
End if
```

# On Startup database method

On Startup database method
Does not require any parameters

---

The **On Startup database method** is called once when you open a database.

This occurs in the following 4D environments:

- 4D in local mode
- 4D in remote mode (on the client side, after the connection has been accepted by 4D Server)
- 4D application compiled and merged with 4D Volume Desktop

**Note**: The **On Startup database method** is NOT invoked by 4D Server.

The **On Startup database method** is automatically invoked by 4D; unlike project methods, you cannot call this database method yourself by programming. You can however execute it from the Method editor. You can also use subroutines.

The **On Startup database method** is the perfect place to:

- Initialize interprocess variables that you will use during the whole working session.
- Start processes automatically when a database is opened.
- Load Preferences or Settings saved for this purpose during the previous working session.
- Prevent the opening of the database if a condition is not met (i.e., missing system resources) by explicitly calling **QUIT 4D**.
- Perform any other actions that you want to be performed automatically each time a database is opened.

However, we strongly recommend that you do NOT launch print jobs from the **On Startup database method**.

## Example

See the example in the **On Exit database method** section.

# On System Event database method

$1 -> On System Event database method

| Parameter | Type | | Description |
|---|---|---|---|
| $1 | Longint | ⇦ | Event code |

## Description

The **On System Event database method** is called each time a system event occurs. This concerns all 4D environments: 4D (all modes) and 4D Server, as well as 4D applications that are compiled and merged with 4D Volume Desktop.

To process an event, you must test the value of the $1 parameter within the method and compare it to one of the following constants, found in the **Database Events** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| On application background move | Longint | 1 | The 4D application moves to the background |
| On application foreground move | Longint | 2 | The 4D application moves to the foreground |

These events are generated when a 4D application changes level, irrespective of the user action initiating this change. For example:

- Clicking the window of the application or of another application,
- selecting it using the **Alt+Tab** (Windows) or **Command+Tab** (Mac OS) keyboard shortcut,
- Selecting the **Hide** command in the dock (Mac OS),
- Clicking the application icon in the dock or task bar,
- Clicking the minimize button of the main window (Windows).

It is absolutely necessary to declare the $1 parameter (longint) in the database method. The structure of the database method code is therefore:

```
 // On System Event database method

C_LONGINT($1)
Case of
   :($1=On application background move)
 //Do something
   :($1=On application foreground move)
 //Do something else
End case
```

# On Web Authentication database method

$1, $2, $3, $4, $5, $6 -> On Web Authentication database method -> $0

| Parameter | Type | | Description |
|---|---|---|---|
| $1 | Text | ⇦ | URL |
| $2 | Text | ⇦ | HTTP header + HTTP body |
| $3 | Text | ⇦ | IP address of browser |
| $4 | Text | ⇦ | IP address of the server |
| $5 | Text | ⇦ | User name |
| $6 | Text | ⇦ | Password |
| $0 | Boolean | ⇨ | True = request accepted, False = request refused |

## Description

The **On Web Authentication database method** is in charge of managing Web server engine access. It is called by 4D or 4D Server when a Web browser request requires the execution of a 4D method on the server (method called using a *4DACTION* URL, a *4DSCRIPT* tag, etc.).

This method receives six Text parameters: $1, $2, $3, $4, $5, and $6, and returns one Boolean parameter, $0. The description of these parameters is as follows:

| Parameters | Type | Description |
|---|---|---|
| $1 | Text | URL |
| $2 | Text | HTTP header + HTTP body (32 KB maximum) |
| $3 | Text | IP address of the Web client (browser) |
| $4 | Text | IP address of the server |
| $5 | Text | User name |
| $6 | Text | Password |
| $0 | Boolean | True = request accepted, False = request rejected |

You must declare these parameters as follows:

```
` On Web Authentication Database Method

C_TEXT($1;$2;$3;$4;$5;$6)
C_BOOLEAN($0)

` Code for the method
```

**Note:** All the On Web Authentication database method's parameters are not necessarily filled in. The information received by the database method depends on the options that you have previously selected in the Database Settings dialog box (please refer to the section **Connection Security**).

- **URL**

  The first parameter (*$1*) is the URL entered by the user in the location area of his or her Web browser, from which the host address has been removed.

  Let's take the example of an Intranet connection. Suppose that the IP address of your 4D Web Server machine is *123.4.567.89*. The following table shows the values of *$1* depending on the URL entered in the Web browser:

  | URL entered in Web browser Location area | Value of parameter $1 |
  |---|---|
  | 123.4.567.89 | / |
  | http://123.4.567.89 | / |
  | 123.4.567.89/Customers | /Customers |
  | http://123.4.567.89/Customers | /Customers |
  | http://123.4.567.89/Customers/Add | /Customers/Add |
  | 123.4.567.89/Do_This/If_OK/Do_That | /Do_This/If_OK/Do_That |

- **Header and Body of the HTTP request**

The second parameter (*$2*) is the header and the body of the HTTP request sent by the Web browser. Note that this information is passed to your **On Web Authentication database method** as it is. Its contents will vary depending on the nature of the Web browser which is attempting the connection.
If your application deals with this information, it is up to you to parse the header and the body.

**Notes:**

- For performance reasons, the size of data passing through the $2 parameter must not exceed 32 KB. Beyond this size, they are truncated by the 4D HTTP server.
- For more information about this parameter, please refer to the description of the **On Web Connection database method**.

- **Web client IP address**

  The $3 parameter receives the IP address of the browser's machine. This information can allow you to distinguish between Intranet and Internet connections.
  **Note:** 4D returns IPv4 addresses in a hybrid IPv6/IPv4 format written with a 96-bit prefix, for example ::ffff:192.168.2.34 for the IPv4 address 192.168.2.34. For more information, refer to the **Support of IPv6** section.

- **Server IP address**

  The $4 parameter receives the IP address used to call the Web server. 4D since version 6.5 allows for multi-homing, which allows you to exploit machines with more than one IP address. For more information, please refer to the section **Web Server Settings**

- **User Name and Password**

  The $5 and $6 parameters receive the user name and password entered by the user in the standard identification dialog box displayed by the browser. This dialog box appears for each connection, if a password management option has been selected in the Database Settings dialog box (see section **Connection Security**).

**Note:** If the user name sent by the browser exists in 4D, the $6 parameter (the user's password) is not returned for security reasons.

- **$0 parameter**

  The **On Web Authentication database method** returns a boolean in $0:
  - If $0 is **True**, the connection is accepted.
  - If $0 is **False**, the connection is refused.

The **On Web Connection database method** is only executed if the connection has been accepted by **On Web Authentication**.

**WARNING:** If no value is set to *$0* or if *$0* is not defined in the **On Web Authentication database method**, the connection is considered as accepted and the **On Web Connection database method** is executed.

**Notes :**

- Do not call any interface elements in the **On Web Authentication database method** (**ALERT**, **DIALOG**, etc.) because otherwise its execution will be interrupted and the connection refused. The same thing will happen if an error occurs during its processing.
- It is possible to prevent execution by *4DACTION* or *4DSCRIPT* for each project method via the "Available through 4D HTML tags and URLs (4DACTION...)" option in the Method properties dialog box. For more information about this point, please refer to the **Connection Security** section.

## On Web Authentication Database Method calls

The **On Web Authentication database method** is automatically called, regardless of the mode, when a request or processing requires the execution of a 4D method. It is also called when the Web server receives an invalid static URL (for example, if the static page requested does not exist).

The **On Web Authentication database method** is therefore called in the following cases:

- when 4D receives a URL beginning with *4DACTION/*
- when 4D receives a URL beginning with *4DCGI/*
- when 4D receives a URL beginning with *4DSYNC/*
- when 4D receives a URL requesting a static page that does not exist
- when 4D receives a root access URL and no home page has been set in the Database Settings or by means of the **WEB SET HOME PAGE** command

- when 4D processes a *4DSCRIPT* tag in a semi-dynamic page
- when 4D processes a *4DLOOP* tag based on a method in a semi-dynamic page.

**Compatibility note:** The database method is also called when 4D receives a URL beginning with *4DMETHOD/*. This URL is obsolete and is only kept for compatibility's sake.

Note that the **On Web Authentication database method** is NOT called when the server receives a URL requesting a valid static page.

## Example 1

Example of the **On Web Authentication database method** in BASIC mode:

```
 `On Web Authentication Database Method
C_TEXT($5;$6;$3;$4)
C_TEXT($user;$password;$BrowserIP;$ServerIP)
C_BOOLEAN($4Duser)
ARRAY TEXT($users;0)
ARRAY LONGINT($nums;0)
C_LONGINT($upos)
C_BOOLEAN($0)

$0:=False

$user:=$5
$password:=$6
$BrowserIP:=$3
$ServerIP:=$4

 `For security reasons, refuse names that contain @
If(WithWildcard($user)|WithWildcard($password))
   $0:=False
 `The WithWildcard method is described below
Else
 `Check to see if it's a 4D user
   GET USER LIST($users;$nums)
   $upos:=Find in array($users;$user)
   If($upos >0)
      $4Duser:=Not(Is user deleted($nums{$upos}))
   Else
      $4Duser:=False
   End if

   If(Not($4Duser))
 `It is not a user defined 4D, look in the table of Web users
      QUERY([WebUsers];[WebUsers]User=$user;*)
      QUERY([WebUsers]; & [WebUsers]Password=$password)
      $0:=(Records in selection([WebUsers])=1)
   Else
      $0:=True
   End if
End if
 `Is this an intranet connection?
If(Substring($BrowserIP;1;7)#"192.100.")
   $0:=False
End if
```

## Example 2

Example of the **On Web Authentication database method** in DIGEST mode:

```
 `On Web Authentication Database Method
C_TEXT($1;$2;$5;$6;$3;$4)
C_TEXT($user)
C_BOOLEAN($0)
$0:=False
$user:=$5
```

```
   `For security reasons, refuse names that contain @
If(WithWildcard($user))
   $0:=False
   `The WithWildcard method is described below
Else
   QUERY([WebUsers];[WebUsers]User=$user)
   If(OK=1)
      $0:=Validate Digest Web Password($user;[WebUsers]password)
   Else
      $0:=False `User does not exist
   End if
End if
```

The ***WithWildcard*** method is as follows:

```
   `WithWildcard Method
   `WithWildcard ( String ) -> Boolean
   `WithWildcard ( Name ) -> Contains a Wilcard character

C_INTEGER($i)
C_BOOLEAN($0)
C_TEXT($1)

$0:=False
For($i;1;Length($1))
   If(Character code(Substring($1;$i;1))=Character code("@"))
      $0:=True
   End if
End for
```

# On Web Close Process database method

On Web Close Process database method
Does not require any parameters

---

The **On Web Close Process database method** is called by the 4D Web server each time a Web session is about to be closed. A session can be closed in the following cases:

- when the maximum number of simultaneous sessions is reached (100 by default, modifiable using the **WEB SET OPTION** command), and 4D needs to create new ones (4D automatically kills the process of the oldest inactive session),
- when the maximum period of inactivity for the session process is reached (480 minutes by default, modifiable using the **WEB SET OPTION** command),
- when the **WEB CLOSE SESSION** command is called.

When this database method is called, the context of the session (variables and selections generated by the user) is still valid. This means that you can save data related to the session in order to be able to use them again subsequently, more specifically using the **On Web Connection Database Method**.

**Note:** In the context of a 4D Mobile session (which can generate several processes), the **On Web Close Process database method** is called for each Web process that is closed, allowing you to save all types of data (variables, selection, etc.) generated by the 4D Mobile session process.

An example of the **On Web Close Process database method** is provided in the **Web Sessions Management** section.

## On Web Connection database method

$1, $2, $3, $4, $5, $6 -> On Web Connection database method

| Parameter | Type | | Description |
|---|---|---|---|
| $1 | Text | ⇐ | URL |
| $2 | Text | ⇐ | HTTP header + HTTP body |
| $3 | Text | ⇐ | IP address of browser |
| $4 | Text | ⇐ | IP address of the server |
| $5 | Text | ⇐ | User name |
| $6 | Text | ⇐ | Password |

---

The **On Web Connection database method** can be called in the following cases:

- the Web server receives a request beginning with the *4DCGI* URL.
- the Web server receives an invalid request.

For more information, refer to the paragraph “On Web Connection Database Method calls” below.

**Compatibility note**: The database method is also called when a context is created in contextual mode (obsolete mode that could be used in converted 4D databases).

The request should have been previously accepted by the **On Web Authentication database method** (if it exists) and the Web server must be launched.

The **On Web Connection database method** receives six text parameters that are passed by 4D. The contents of these parameters are as follows:

| Parameters | Type | Description |
|---|---|---|
| $1 | Text | URL |
| $2 | Text | HTTP header + HTTP body (up to 32 kb limit) |
| $3 | Text | IP address of the Web client (browser) |
| $4 | Text | IP address of the server |
| $5 | Text | User name |
| $6 | Text | Password |

You must declare these parameters as follows:

```
` On Web Connection Database Method

C_TEXT($1;$2;$3;$4;$5;$6)

` Code for the method
```

- **URL extra data**

  The first parameter (*$1*) is the URL entered by the user in the location area of his or her Web browser, from which the host address has been removed.
  Let’s take the example of an Intranet connection. Suppose that the IP address of your 4D Web Server machine is *123.4.567.89*. The following table shows the values of *$1* depending on the URL entered in the Web browser:

| URL entered in Web browser Location area | Value of parameter $1 |
|---|---|
| 123.4.567.89 | / |
| http://123.4.567.89 | / |
| 123.4.567.89/Customers | /Customers |
| http://123.4.567.89/Customers | /Customers |
| http://123.4.567.89/Customers/Add | /Customers/Add |
| 123.4.567.89/Do_This/If_OK/Do_That | /Do_This/If_OK/Do_That |

  Note that you are free to use this parameter at your convenience. 4D simply ignores the value passed beyond the host

part of the URL.
For example, you can establish a convention where the value *"/Customers/Add"* means "go directly to add a new record in the *[Customers]* table." By supplying the Web users of your database with a list of possible values and/or default bookmarks, you can provide shortcuts to the different parts of your application. This way, Web users can quickly access resources of your Web site without going through the whole navigation path each time they make a new connection to your database.

**WARNING:** In order to prevent a user from reentering a database with a bookmark created during a previous session, 4D intercepts any URL that corresponds to one of the standard 4D URLs.

- **Header of the HTTP request followed by the HTTP body**
  The second parameter (*$2*) is the header and the body of the HTTP request sent by the Web browser. Note that this information is passed to your **On Web Connection database method** as it is. Its contents will vary depending on the nature of the Web browser which is attempting the connection.
  With Safari running on Mac OS, you may receive a header similar to this:

```
GET /favicon.ico HTTP/1.1
Referer: http://123.45.67.89/4dcgi/test
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; fr-fr) AppleWebKit/523.10.3 (KHTML, like Gecko) Version/3.0.4
Safari/523.10
Cache-Control: max-age=0
Accept: */*
Accept-Language: fr-fr
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: 123.45.67.89
```

  With Microsoft Internet Explorer 8 running on Windows, you may receive a header similar to this:

```
GET / HTTP/1.1
Accept: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/pjpeg, application/x-ms-xbap,
application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, */*
Accept-Language: fr-FR
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR
3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C)
Accept-Encoding: gzip, deflate
Host: 123.45.67.89
Connection: Keep-Alive
```

  If your application deals with this information, it is up to you to parse the header and the body.
  **Note:** For performance reasons, the size of these data cannot be more than 32 KB. Beyond this, they are truncated by the 4D HTTP server.

- **IP address of the Web client**
  The *$3* parameter receives the IP address of the browser's machine. This information can allow you to distinguish between Intranet and Internet connections.
  **Note:** 4D returns IPv4 addresses in a hybrid IPv6/IPv4 format written with a 96-bit prefix, for example ::ffff:192.168.2.34 for the IPv4 address 192.168.2.34. For more information, refer to the **Support of IPv6** section.

- **IP address of the server**
  The *$4* parameter receives the IP address to which the HTTP request was sent. 4D allows for multi-homing, which allows you to exploit machines with more than one IP address. For more information, please refer to the section **Web Server Settings**.

- **User Name and Password**
  The *$5* and *$6* parameters receive the user name and password entered by the user in the standard identification dialog box displayed by the browser. This dialog box appears for each connection, if the **Use Passwords** option has been selected in the Database Settings dialog box (see section **Connection Security**).

**Note:** If the user name sent by the browser exists in 4D, the *$6* parameter (the user's password) is not returned for security reasons.

## On Web Connection Database Method Calls

The **On Web Connection database method** can be used as the entry point for the 4D Web server, either using the special *4DCGI* URL, or using customized command URLs.

**Warning:** Calling a 4D command that displays an interface element (**DIALOG**, **ALERT**, etc.) ends the method processing.

The **On Web Connection database method** is therefore called in the following cases:

- When 4D receives the */4DCGI* URL. The database method is called with the */4DCGI/<action>* URL in *$1*.
- When a Web page is called with a URL of type *<path>/<file>* is not found. The database method is called with the URL (*).
- When a Web page is called with a URL of type *<file>/* and no home page has been defined by default. The database method is called with the URL (*).

(*) In this particular cases, the URL received in *$1* does NOT start with the "/" character.

# Date and Time

- Add to date
- Current date
- Current time
- Date
- Day number
- Day of
- Milliseconds
- Month of
- SET DEFAULT CENTURY
- Tickcount
- Time
- Time string
- Year of

# Add to date

Add to date ( date ; years ; months ; days ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| date | Date | ➡ | Date to which to add days, months, and years |
| years | Longint | ➡ | Number of years to add to the date |
| months | Longint | ➡ | Number of months to add to the date |
| days | Longint | ➡ | Number of days to add to the date |
| Function result | Date | ⮌ | Resulting date |

## Description

The **Add to date** command adds *years*, *months,* and *days* to the date you pass in *aDate*, then returns the result.

Although you can use the **Date Operators** to add days to a date, **Add to date** allows you to quickly add months and years without having to deal with the number of days per month or leap years (as you would when using the + date operator).

## Example

```
 ` This line calculates the date in one year, same day
$vdInOneYear:=Add to date(Current date;1;0;0)

 ` This line calculates the date next month, same day
$vdNextMonth:=Add to date(Current date;0;1;0)

 ` This line does the same thing as $vdTomorrow:=Current date+1
$vdTomorrow:=Add to date(Current date;0;0;1)
```

# Current date

Current date {( * )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | Returns the current date from the server |
| Function result | Date | ⇐ | Current date |

## Description

The **Current date** command returns the current date as kept by the system clock.

**4D Server:** If you use the asterisk (*) parameter when executing this function on a 4D Client machine, it returns the current date from the server.

## Example 1

The following example displays an alert box containing the current date:

```
ALERT("The date is "+String(Current date)+".")
```

## Example 2

If you write an application for the international market, you may need to know if the version of 4D that you run works with dates formatted as MM/DD/YYYY (US version) or DD/MM/YYYY (French version). This is useful to know for customizing data entry fields.

The following project method allows you to do so:

```
  ` Sys date format global function
  ` Sys date format -> String
  ` Sys date format -> Default 4D data format

C_STRING(31;$0;$vsDate;$vsMDY;$vsMonth;$vsDay;$vsYear)
C_LONGINT($1;$vlPos)
C_DATE($vdDate)

  ` Get a Date value where the month, day and year values are all different
$vdDate:=Current date
Repeat
   $vsMonth:=String(Month of($vdDate))
   $vsDay:=String(Day of($vdDate))
   $vsYear:=String(Year of($vdDate)%100)
   If(($vsMonth=$vsDay)|($vsMonth=$vsYear)|($vsDay=$vsYear))
      vOK:=0
      $vdDate:=$vdDate+1
   Else
      vOK:=1
   End if
Until(vOK=1)
$0:="" ` Initialize function result
$vsDate:=String($vdDate)
$vlPos:=Position("/";$vsDate) ` Find the first / separator in the string ..
$vsMDY:=Substring($vsDate;1;$vlPos-1) ` Extract the first digits from the date
$vsDate:=Substring($vsDate;$vlPos+1) ` Eliminate the first digits as well as the first / separator
Case of
   :($vsMDY=$vsMonth) ` The digits express the month
      $0:="MM"
   :($vsMDY=$vsDay) ` The digits express the day
      $0:="DD"
   :($vsMDY=$vsYear) ` The digits express the year
      $0:="YYYY"
```

```
End case
$0:=$0+"/" ` Start building the function result
$vlPos:=Position("/";$vsDate) ` Find the second separator in the string ../..
$vsMDY:=Substring($vsDate;1;$vlPos-1) ` Extract the next digits from the date
$vsDate:=Substring($vsDate;$vlPos+1) ` Reduce the string to the last digits from the date
Case of
   :($vsMDY=$vsMonth) ` The digits express the month
      $0:=$0+"MM"
   :($vsMDY=$vsDay) ` The digits express the day
      $0:=$0+"DD"
   :($vsMDY=$vsYear) ` The digits express the year
      $0:=$0+"YYYY"
End case
$0:=$0+"/" ` Pursue building the function result
Case of
   :($vsDate=$vsMonth) ` The digits express the month
      $0:=$0+"MM"
   :($vsDate=$vsDay) ` The digits express the day
      $0:=$0+"DD"
   :($vsDate=$vsYear) ` The digits express the year
      $0:=$0+"YYYY"
End case
 ` At this point $0 is equal to MM/DD/YYYY or DD/MM/YYYY or...
```

# Current time

Current time {( * )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | Returns the current time from the server |
| Function result | Time | ⇒ | Current time |

## Description

The **Current time** command returns the current time from the system clock.

The current time is always between *00:00:00* and *23:59:59*. Use **String** or **Time string** to obtain the string form of the time expression returned by **Current time**.

**4D Server:** If you use the asterisk (*) parameter when executing this function on a 4D Client machine, it returns the current time from the server.

## Example 1

The following example shows you how to time the length of an operation. Here, ***LongOperation*** is a method that needs to be timed:

```
$vhStartTime:=Current time ` Save the start time
LongOperation ` Perform the operation
ALERT("The operation took "+String(Current time-$vhStartTime)) ` Display how long it took
```

## Example 2

The following example extracts the hours, minutes, and seconds from the current time:

```
$vhNow:=Current time
ALERT("Current hour is: "+String($vhNow¥3600))
ALERT("Current minute is: "+String(($vhNow¥60)%60))
ALERT("Current second is: "+String($vhNow%60))
```

# Date

Date ( dateString ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| dateString | String | ➡ | String representing the date to be returned |
| Function result | Date | ⮌ | Date |

## Description

The **Date** command evaluates *dateString* and returns a date.

The *dateString* parameter format must follow either the ISO date format or the regional settings defined at the system level.

**ISO date format**

The string must be formatted as follows: "YYYY-MM-DD**T**HH:MM:SS", for example "2013-11-20T10:20:00". In this case, **Date** evaluates the *dateString* parameter correctly, whatever the current language settings. Decimal seconds, preceded by a period, are supported (e.g.: "2013-11-20T10:20:00.9854").

If the *dateString* format does not precisely fit this ISO schema, then the date is evaluated as a short date format based on the regional settings of the system.

**Note :** Starting with 4D v14, it is recommended to use the "YYYY-MM-DDTHH:MM:SS**Z**" format, compliant with the ISO standard and allowing to express the time zone.

**Regional settings**

When the *dateString* does not match the ISO format, the regional settings defined in the operating system for a short date are used for the evaluation. For example, in the US version of 4D, by default the date must be in the order MM/DD/YY (month, day, year). The month and day can be one or two digits. The year can be two or four digits. If the year is two digits, then **Date** considers whether the date belongs to the 21st or 20th century based on the value entered. By default, the pivotal value is 30:

- If the value is greater than or equal to 30, 4D considers the century to be the 20th and adds 19 to the beginning of the value.
- If the value is less than 30, 4D considers the century to be the 21st and adds 20 to the beginning of the value.

This mechanism can be configured using the **SET DEFAULT CENTURY** command.

The following characters are valid date separators: slash (/), space, period (.), comma (,), and dash (-).

If you pass an invalid date (such as "13/35/94" or "aa/12/94") in *dateString*, **Date** returns a null date (!00/00/00!). It is your responsibility to verify that *dateString* is a valid date.

## Example 1

The following example uses a request box to prompt the user for a date. The string entered by the user is converted to a date and stored in the *reqDate* variable:

```
vdRequestedDate:=Date(Request("Please enter the date:";String(Current date)))
If(OK=1)
   ` Do something with the date now stored in vdRequestedDate
End if
```

## Example 2

The following example returns the string "12/25/94" as a date:

```
vdDate:=Date("12/25/94")
```

## Example 3

Date evaluation based on a date in ISO format:

```
$vtDateISO:="2013-06-05T20:00:00"
$vDate:=Date($vtDateISO)
 //$vDate represents June 5th, 2013 regardless of the system language
```

# Day number

Day number ( aDate ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aDate | Date | ➡ | Date for which to return the number |
| Function result | Longint | ⮌ | Number representing the weekday on which date falls |

## Description

The **Day number** command returns a number representing the weekday on which *aDate* falls.

**Note: Day number** returns *2* for null dates.

4D provides the following predefined constants, found in the "**Days and Months**" theme:

| Constant | Type | Value |
|---|---|---|
| Sunday | Longint | 1 |
| Monday | Longint | 2 |
| Tuesday | Longint | 3 |
| Wednesday | Longint | 4 |
| Thursday | Longint | 5 |
| Friday | Longint | 6 |
| Saturday | Longint | 7 |

**Note: Day number** returns a value between 1 and 7. To get the day number within the month for a date, use the command **Day of**.

## Example

The following example is a function that returns the current day as a string:

```
$viDay :=Day number(Current date) ` $viDay gets the current day number
Case of
   :($viDay =1)
      $0:="Sunday"
   :($viDay =2)
      $0:="Monday"
   :($viDay =3)
      $0:="Tuesday"
   :($viDay =4)
      $0:="Wednesday"
   :($viDay =5)
      $0:="Thursday"
   :($viDay =6)
      $0:="Friday"
   :($viDay =7)
      $0:="Saturday"
End case
```

# Day of

Day of ( date ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| date | Date | ➡ | Date for which to return the day |
| Function result | Longint | ⮌ | Day of the month of date |

## Description

The **Day of** command returns the day of the month of *aDate*.

**Note: Day of** returns a value between 1 and 31. To get the day of the week for a date, use the command **Day number**.

## Example 1

The following example illustrates the use of **Day of**. The results are assigned to the variable *vResult*. The comments describe what is put in *vResult*:

```
vResult:=Day of(!12/25/92!) ` vResult gets 25
vResult:=Day of(Current date) ` vResult gets day of current date
```

## Example 2

See the example for the **Current date** command.

# Milliseconds

Milliseconds -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | | Number of milliseconds elasped since the machine was started |

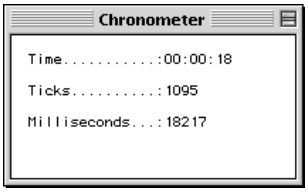
## Description

**Milliseconds** returns the number of milliseconds (1000th of a second) elapsed since the machine was started.

## Example

The following code displays the “Chronometer” window for one minute:

```
Open window(100;100;300;200;0;"Chronometer")
$vhTimeStart:=Current time
$vlTicksStart:=Tickcount
$vrMillisecondsStart:=Milliseconds
Repeat
   GOTO XY(2;1)
   MESSAGE("Time..........:"+String(Current time-$vhTimeStart))
   GOTO XY(2;3)
   MESSAGE("Ticks.........:"+String(Tickcount-$vlTicksStart))
   GOTO XY(2;5)
   MESSAGE("Milliseconds...:"+String(Milliseconds-$vrMillisecondsStart))
Until((Current time-$vhTimeStart)>=?00:01:00?)
CLOSE WINDOW
```



# Month of

Month of ( aDate ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aDate | Date | ⇒ | Date for which to return the month |
| Function result | Longint | ⇐ | Number indicating the month of date |

## Description

The **Month of** command returns the month of *aDate*.

**Note: Month of** returns the number of the month, not the name (see Example 1).

To compare the value returned by this function, 4D provides the following predefined constants, found in the "**Days and Months**" theme:

| Constant | Type | Value |
|---|---|---|
| January | Longint | 1 |
| February | Longint | 2 |
| March | Longint | 3 |
| April | Longint | 4 |
| May | Longint | 5 |
| June | Longint | 6 |
| July | Longint | 7 |
| August | Longint | 8 |
| September | Longint | 9 |
| October | Longint | 10 |
| November | Longint | 11 |
| December | Longint | 12 |

## Example 1

The following example illustrates the use of **Month of**. The results are assigned to the variable *vResult*. The comments describe what is put in *vResult*:

```
vResult:=Month of(!12/25/92!) ` vResult gets 12
vResult:=Month of(Current date) ` vResult gets month of current date
```

## Example 2

See example for the **Current date** command.

# SET DEFAULT CENTURY

SET DEFAULT CENTURY ( century {; pivotYear} )

| Parameter | Type | | Description |
|---|---|---|---|
| century | Longint | ⇒ | Default century (minus one) for entry of date with two-digit year |
| pivotYear | Longint | ⇒ | Pivot year for entry of date with two-digit year |

## Description

The **SET DEFAULT CENTURY** command specifies the default century and the pivot year used by 4D when you enter a date with only two digits for the year.

The pivot year value defines the way 4D will interpret data entry of a date with a two-digit year:

- If the year is greater than or equal to the pivot year, 4D uses the current default century.
- If the year is less than the pivot year, 4D uses the next century (relative to the current default).

By default, 4D sets the century to be the 20th century and uses 30 as pivot year. For example:

- 01/25/97 means January 25, 1997.
- 01/25/30 means January 25, 1930.
- 01/25/29 means January 25, 2029.
- 01/25/07 means January 25, 2007.

To change this default, execute the command. The effect of the command is immediate. You can pass a new default century only, or a new default century and a pivot year.

If you pass only a new default century minus one in *century*, 4D will interpret data entry of a date with a two-digit year as belonging to this century.
For example, after the call:

```
SET DEFAULT CENTURY(20) ` Switch to 21st century for default century
```

- 01/25/97 means January 25, 2097
- 01/25/07 means January 25, 2007

In addition, you can specify the optional *pivotYear* parameter.
For example, after this call, in which the pivot year is 1995:

```
SET DEFAULT CENTURY(19;95) ` Switch to 21st century for default century if year is less than 95
```

- 01/25/97 means January 25, 1997
- 01/25/07 means January 25, 2007

**Note:** This command only affects how 4D interprets dates entered with a two-digit year.
In all cases:

- 01/25/1997 means January 25, 1997
- 01/25/2097 means January 25, 2097
- 01/25/1907 means January 25, 1907
- 01/25/2007 means January 25, 2007

This command only affects data entry. It has no effect on date storage, computation, and so on.

# Tickcount

Tickcount -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Number of ticks (60th of a second) elapsed since the machine was started |

## Description

**Tickcount** returns the number of ticks (60th of a second) elapsed since the machine was started.

**Note: Tickcount** returns a value of type Long Integer.

## Example

See example for the command **Milliseconds**.

# Time

Time ( timeValue ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| timeValue | String, Longint | ⇒ | Value to return as a time |
| Function result | Time | ⇐ | Time specified by timeValue |

## Description

The **Time** command returns a time expression equivalent to the time specified in the *timeValue* parameter.

The *timeValue* parameter can contain either:

- a string containing a time expressed in one of the standard time formats of 4D corresponding to the language of your system (for more information, refer to the description of the **String** command).
- a longint that represents the number of seconds elapsed since 00:00:00.

## Example 1

The following example displays an alert box with the message “1:00 P.M. = 13 hours 0 minute”:

```
ALERT("1:00 P.M. = "+String(Time("13:00:00");Hour Min))
```

## Example 2

You can express any numerical value as a time:

```
vTime:=Time(10000)
 //vTime is 02:46:40
vTime2:=Time((60*60)+(20*60)+5200)
 //vTime2 is 02:46:40
```

# Time string

Time string ( seconds ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| seconds | Longint, Time | ⇒ | Seconds from midnight |
| Function result | String | ⇐ | Time as a string in 24-hour format |

## Description

The **Time string** command returns the string form of the time expression you pass in *seconds*.

The string is in the *HH:MM:SS* format.

If you go beyond the number of seconds in a day (86,400), **Time string** continues to add hours, minutes, and seconds. For example, **Time string** (86401) returns 24:00:01.

**Note:** If you need the string form of a time expression in a variety of formats, use **String**.

## Example

The following example displays an alert box with the message, “46800 seconds is 13:00:00.”

```
ALERT("46800 seconds is "+Time string(46800))
```

# Year of

Year of ( date ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| date | Date | ⇒ | Date for which to return the year |
| Function result | Longint | ⇐ | Number indicating the year of date |

## Description

The **Year of** command returns the year of *aDate*.

## Example 1

The following example illustrates the use of **Year of**. The results are assigned to the variable *vResult*.

```
vResult:=Year of(!12/25/92!) ` vResult gets 1992
vResult:=Year of(!12/25/1992!) ` vResult gets 1992
vResult:=Year of(!12/25/1892!) ` vResult gets 1892
vResult:=Year of(!12/25/2092!) ` vResult gets 2092
vResult:=Year of(Current date) ` vResult gets year of current date
```

## Example 2

See example for the command **Current date**.

# Design Object Access

- Design Object Access Commands
- Current method path
- FORM GET NAMES
- METHOD Get attribute
- METHOD GET ATTRIBUTES
- METHOD GET CODE
- METHOD GET COMMENTS
- METHOD GET FOLDERS
- METHOD GET MODIFICATION DATE
- METHOD GET NAMES
- METHOD Get path
- METHOD GET PATHS
- METHOD GET PATHS FORM
- METHOD OPEN PATH
- METHOD RESOLVE PATH
- METHOD SET ACCESS MODE
- METHOD SET ATTRIBUTE
- METHOD SET ATTRIBUTES
- METHOD SET CODE
- METHOD SET COMMENTS

# Design Object Access Commands

With 4D, you can access the contents of methods in your applications by programming. This source toolkit facilitates the integration into your applications of code control tools and more particularly version control systems (VCS). It also lets you implement advanced systems for code documentation, for building a custom explorer or for organizing scheduled backups of the code saved as disk files.

The following principles are implemented:

- Each method and form in a 4D application has its own address in the form of a pathname. For example, the trigger method for table 1 can be found at "[trigger]/table_1". Each object pathname is unique in an application.
  **Note:** To ensure the uniqueness of pathnames, 4D no longer lets you create objects with the same name on different form pages. In databases converted from versions prior to 4D v13, the MSC detects these duplicate names.
- You can access objects in the 4D application using the commands of this theme, for example **METHOD GET NAMES** or **METHOD GET PATHS**.
- Most of the commands in this theme work in both interpreted and compiled mode. However, commands that modify properties or access contents executable from methods can only be used in interpreted mode (see the table below).
- You can use all the commands of this theme with 4D in local or remote mode.
  However, keep in mind that you cannot use certain commands in compiled mode: the purpose of this theme is to create custom development support tools. You must not use these commands to dynamically change the functioning of a database that is running. For example, you cannot use **METHOD SET ATTRIBUTE** to change a method attribute according to the status of the current user.
- When a command of this theme is called from a component by default it accesses the component objects. In this case, to access objects of the host database, you just pass a * as the last parameter. Note that in this context, this syntax is only possible for commands that modify objects (such as **METHOD SET ATTRIBUTE**), because components are always executed in read-only mode.

## Use in compiled mode

For reasons related to the principle of the compilation process, only certain commands in this theme can be used in compiled mode. The following table indicates the available of the commands in compiled mode:

| Command | Can be used in compiled mode |
|---|---|
| Current method path | Yes |
| FORM GET NAMES | Yes |
| METHOD Get attribute | Yes |
| METHOD GET ATTRIBUTES | Yes |
| METHOD GET CODE | No (*) |
| METHOD GET COMMENTS | Yes |
| METHOD GET FOLDERS | Yes |
| METHOD GET MODIFICATION DATE | Yes |
| METHOD GET NAMES | Yes |
| METHOD Get path | Yes |
| METHOD GET PATHS | Yes |
| METHOD GET PATHS FORM | Yes |
| METHOD OPEN PATH | No (*) |
| METHOD RESOLVE PATH | Yes |
| METHOD SET ACCESS MODE | Yes |
| METHOD SET ATTRIBUTE | No (*) |
| METHOD SET ATTRIBUTES | No (*) |
| METHOD SET CODE | No (*) |
| METHOD SET COMMENTS | No (*) |

(*) The error -9762 "The command cannot be executed in a compiled database." is generated when the command is executed in compiled mode.

## Creation of pathnames

By default, no file is created on disk by 4D. However, pathnames generated for objects are compatible with the file management of the operating system; they can generate files directly on disk using your own import/export methods.

More specifically, forbidden characters such as ":" are encoded in method names. Generated files may be integrated automatically in a version control system.
Here are the encoded characters:

| Character | Encoding |
|---|---|
| " | %22 |
| * | %2A |
| / | %2F |
| : | %3A |
| < | %3C |
| > | %3E |
| ? | %3F |
| \| | %7C |
| ¥ | %5C |
| % | %25 |

**Examples:**
*Form?1* is encoded *Form%3F1*
*Button/1* is encoded *Button%2F1*

# Current method path

Current method path -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Text | ➲ | Full internal pathname of the method being executed |

## Description

The **Current method path** command returns the internal pathname of the database method, trigger, project method, form method or object method being executed.

**Note:** In the context of 4D macro-commands, the <*method_path*> tag is replaced in the code by the full pathname of the method being executed.

# FORM GET NAMES

FORM GET NAMES ( {aTable ;} arrNames {; filter {; marker}}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table reference |
| arrNames | Text array | ⇐ | Array of form names |
| filter | Text | ⇒ | Name filter |
| marker | Longint | ⇒ | Marker for minimum version to return |
| | | ⇐ | New value |
| * | Operator | ⇒ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **FORM GET NAMES** command fills the *arrNames* array with the names of forms in the application.

If you pass the *aTable* parameter, the command returns the names of the table forms associated with this table. If you omit this parameter, the command returns the names of the database project forms.

You can limit this list of forms by passing a comparison string in the *filter* parameter: in this case, only forms whose names match the filter are returned. You can use the @ character in order to specify "starts with", "ends with" or "contains" type filters. If you pass an empty string, the *filter* parameter is ignored.

You can also limit the list of forms using the optional *marker* parameter, which allows you to limit forms returned in *arrNames* to those that were modified after a given time. As part of a version control system, this parameter lets you update only forms that were modified since the last backup.
This principle works as follows: 4D internally maintains a counter of database resource modifications. Since forms are resources, each time a form is created or saved, this counter is incremented. If you pass the *marker* parameter, the command returns, in *arrNames*, only forms corresponding to marker values that are greater than or equal to that of the *marker*. In addition, if you pass a variable in *marker*, the command returns the new value of the modification counter, i.e., the highest one, in this variable. You can then save this value and use it in the next call of the **FORM GET NAMES** command to retrieve only new or modified forms.

If the command is executed from a component, it returns by default the names of the component project forms. If you pass the * parameter, the array contains the forms of the host database.

**Note:** Forms placed in the trash are not listed.

## Example

Examples of typical use:

```
 // List all the project forms of the database
FORM GET NAMES(arr_Names)

 // List forms of the [Employees] table
FORM GET NAMES([Employees] ;arr_Names)

 // List "input" forms of the [Employees] table
FORM GET NAMES([Employees] ;arr_Names;"input_@")

 // List specific project forms of the database
FORM GET NAMES(arr_Names;"dialogue_@")

 // List all project forms in database that were modified since the last synchronization
 // vMarker contains a numeric value
FORM GET NAMES(arr_Names;"";vMarker)

 // List table forms from a component
 // A pointer is necessary because the table name is unknown
FORM GET NAMES(tablePtr->;arr_Names;*)
```

# METHOD Get attribute

METHOD Get attribute ( path ; attribType {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text | ⇒ | Path of project method |
| attribType | Longint | ⇒ | Type of attribute to get |
| * | Operator | ⇒ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |
| Function result | Boolean | ⇐ | True = attribute selected; otherwise False |

## Description

The **METHOD Get attribute** command returns the value of the *attribType* attribute for the project method designated by the *path* parameter. This command only works with project methods. If you pass an invalid *path*, an error is generated.

In the *attribType* parameter, pass a value indicating the type of attribute to get. You can use the following constants, found in the **Design Object Access** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Attribute executed on server | Longint | 8 | Corresponds to the "Execute on server" option |
| Attribute invisible | Longint | 1 | Corresponds to the "Invisible" option |
| Attribute published SOAP | Longint | 3 | Corresponds to the "Offered as a Web Service" option |
| Attribute published SQL | Longint | 7 | Corresponds to the "Available through SQL" option |
| Attribute published Web | Longint | 2 | Corresponds to the "Available through 4D HTML tags and URLs (4DACTION...)" option |
| Attribute published WSDL | Longint | 4 | Corresponds to the "Published in WSDL" option |
| Attribute shared | Longint | 5 | Corresponds to the "Shared by components and host database" option |

If the command is executed from a component, it applies by default to the component methods. If you pass the * parameter, it accesses the methods of the host database.

The command returns **True** when an attribute is selected and **False** if it is deselected.

# METHOD GET ATTRIBUTES

METHOD GET ATTRIBUTES ( path ; attributes {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text, Text array | ⇒ | Method path(s) |
| attributes | Object, Object array | ⇐ | Attribute(s) for selected method(s) |
| * | Operator | ⇒ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD GET ATTRIBUTES** command returns, in *attributes*, the current value of all attributes for the method(s) specified in the *path* parameter.

This command only works with project methods. If you pass an invalid *path*, an error is generated.

In *path*, you can pass either a text containing a method path, or a text array containing an array of paths. You will need to pass the same kind of parameter (variable or array) in *attributes* in order to get the appropriate attributes.

In *attributes*, you pass an object or an array of objects, depending on the kind of parameter passed in *path*. All the attributes for the method(s) are returned as object properties, with "True"/"False" values for Boolean attributes, and text or additional values if necessary (for example, "scope":"table" for the 4D Mobile property).

If the command is executed from a component, by default it applies to the component methods. If you pass the * parameter, it accesses the methods of the host database.

**Note:** The existing **METHOD Get attribute** command is still supported but since it can only return Boolean values, it cannot be used for extended attributes such as 4D Mobile properties.

## Example

You want to get the attributes of the *sendMail* project method. You can write:

```
C_OBJECT($att)
METHOD GET ATTRIBUTES("sendMail";$att)
```

After execution, $att contains, for example:

```
{    "invisible":false,    "preemptive":"capable",    "publishedWeb":false,    "publishedSoap":false,    "publishedWsdl":false,
"shared":false,    "publishedSql":false,    "executedOnServer":false,    "published4DMobile":{        "scope":"table",        "table":"Table_1"
    } }
```

# METHOD GET CODE

METHOD GET CODE ( path ; code {; option} {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text, Text array | ➡ | Text or Text array containing one or more method path(s) |
| code | Text, Text array | ⬅ | Code of designated method(s) |
| option | Longint | ➡ | 0 or omitted = simple export (without tokens), 1 = export with tokens |
| * | Operator | ➡ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD GET CODE** command returns, in the *code* parameter, the contents of the method(s) designated by the *path* parameter. This command can return the code of all types of methods: database methods, triggers, project methods, form methods and object methods.

You can use two types of syntaxes, based either on text arrays, or text variables:

```
C_TEXT(tVpath) // text variables
C_TEXT(tVcode)
METHOD GET CODE(tVpath;tVcode) // code of a single method
```

```
ARRAY TEXT(arrPaths;0) // text arrays
ARRAY TEXT(arrCodes;0)
METHOD GET CODE(arrPaths;arrCodes) // code of several methods
```

You cannot mix the two syntaxes.

If you pass an invalid pathname, the *code* parameter is left empty and an error is generated.

In the text of the *code* generated by this command:

- Command names are written in English for all versions of 4D, except when you use a French version and check the "Use regional system settings" preference (see **Methods Page**). When you use the *option* parameter, the code can contain language tokens in order to make it independent from the 4D programming language and version (see below).
- To increase code readability, text is indented with tab characters based on programming structures, like in the Method editor.
- A line is added in the header of the code generated containing metadata used when importing code, for example:

  ```
  // %attributes = {"lang":"en","invisible":true,"folder":"Web3"}
  ```

  During an import, this line is not imported, it is only used to set the corresponding attributes (attributes that are not specified are reset to their default value). The "lang" attribute sets the export language and prevents an import into an application in a different language (in this case, an error is generated). The "folder" attribute contains the name of the method's parent folder; it is not shown when the method does not have a parent folder.
  Additional attributes can be defined. For more information, refer to the description of the **METHOD SET ATTRIBUTES** command.

The *option* parameter allows you to select the code export mode with respect to the tokenized language elements of the method(s):

- If you pass 0 or omit the *option* parameter, the method code is exported without tokens, i.e. just like it is displayed in the Method editor.
- If you pass 1 or the Code with tokens constant, the method code is exported with tokens, i.e. tokenized elements are followed by their internal reference in the *code* exported contents. For example, the expression "**String**(a)" is exported "**String**:C10(a)", where "C10" is the internal reference of the **String** command.

Tokenized language elements include:

- 4D commands and constants,
- Table and field names,
- 4D plug-in commands.

Code exported with tokens is independent from any subsequent renaming of language elements. Thanks to tokens, code provided as text will always be interpreted correctly by 4D, whether by means of the **METHOD SET CODE** command or even by copy/paste. For more information about the syntax of 4D tokens, please refer to **Using tokens in formulas**.

If the command is executed from a component, it applies by default to the component methods. If you pass the * parameter, it accesses the methods of the host database.

## Example 1

Refer to the example of the **METHOD SET CODE** command.

## Example 2

This example illustrates the effect of the *option* parameter.

You want to export the following "simple_init" method:

```
Case of
   :(Form event=On Load)
      ALL RECORDS([Customer])
End case
```

If you execute the following code:

```
C_TEXT($code)
C_TEXT($contents)
$code:=METHOD Get path(Path project method;"simple_init")
METHOD GET CODE($code;$contents;0) //no tokens
TEXT TO DOCUMENT("simple_init.txt";$contents)
```

The resulting document will contain:

```
  //%attributes = {"lang":"en"} comment added and reserved by 4D
Case of
    : (Form event=On Load)
        ALL RECORDS([Customer])
End case
```

If you execute the following code:

```
C_TEXT($code)
C_TEXT($contents)
$code:=METHOD Get path(Path project method;"simple_init")
METHOD GET CODE($code;$contents;Code with tokens) //use tokens
TEXT TO DOCUMENT("simple_init.txt";$contents)
```

The resulting document will contain:

```
  //%attributes = {"lang":"en"} comment added and reserved by 4D
Case of
    : (Form event:C388=On Load:K2:1)
        ALL RECORDS:C47([Customer:1])
End case
```

# METHOD GET COMMENTS

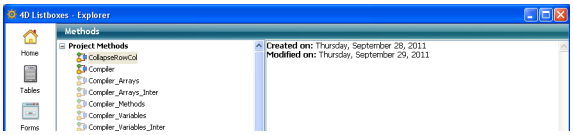
METHOD GET COMMENTS ( path ; comments {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text, Text array | ➡ | Text or Text array containing one or more method path(s) |
| comments | Text, Text array | ⬅ | Comments of designated method(s) |
| * | Operator | ➡ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD GET COMMENTS** command returns, in the *comments* parameter, the comments of the method(s) designated by the *path* parameter.

The comments retrieved by this command are those specified in the 4D Explorer (not to be confused with lines of comments in the code that are retrieved using **METHOD GET CODE**):




These comments can only be generated for triggers, project methods and form methods. They contain styled text.

**Note:** Forms and form methods share the same comments.

You can use two types of syntaxes, based either on text arrays, or text variables:

```
C_TEXT(tVpath) // text variables
C_TEXT(tVcomments)
METHOD GET COMMENTS(tVpath;tVcomments) // comments of a single method
```

```
ARRAY TEXT(arrPaths;0) // text arrays
ARRAY TEXT(arrComments;0)
METHOD GET COMMENTS(arrPaths;arrComments) // comments of several methods
```

You cannot mix the two syntaxes.

If the command is executed from a component, it applies by default to the component methods. If you pass the * parameter, it accesses the methods of the host database.

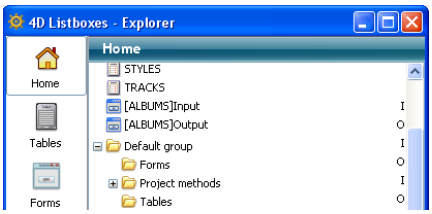
# METHOD GET FOLDERS

METHOD GET FOLDERS ( arrNames {; filter}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrNames | Text array | ⇦ | Array of Home page folder names |
| filter | Text | ⇨ | Name filter |
| * | Operator | ⇨ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD GET FOLDERS** command returns, in the *arrNames* array, the names of folders created on the Home page of the 4D Explorer:




Since folder names must be unique, the hierarchy is not returned in this array.

You can limit this list of folders by passing a comparison string in the *filter* parameter: in this case, only folders whose names match the filter are returned. You can use the @ character in order to specify "starts with", "ends with" or "contains" type filters. If you pass an empty string, the *filter* parameter is ignored.

If the command is executed from a component, it returns by default the paths of the component methods. If you pass the * parameter, the array contains the paths of the methods of the host database.

# METHOD GET MODIFICATION DATE

METHOD GET MODIFICATION DATE ( path ; modDate ; modTime {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text, Text array | ➡ | Text or Text array containing one or more method path(s) |
| modDate | Date, Date array | ⬅ | Method modification date(s) |
| modTime | Time, Longint array | ⬅ | Method modification time(s) |
| * | Operator | ➡ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD GET MODIFICATION DATE** command returns, in the *modDate* and *modTime* parameters, the dates and times of the last modification of the method(s) designated by the *path* parameter.

You can use two types of syntaxes, based either on arrays or variables:

```
C_TEXT(tVpath) // variables
C_DATE(vDate)
C_TIME(vTime)
METHOD GET MODIFICATION DATE(tVpath;vDate;vTime) // date and time of a single method
```

```
ARRAY TEXT(arrPaths;0) // arrays
ARRAY DATE(arrDates;0)
ARRAY LONGINT(arrTimes;0)
METHOD GET MODIFICATION DATE(arrPaths;arrDates;arrTimes) // dates and times of several methods
```

You cannot mix the two syntaxes.

If the command is executed from a component, it applies by default to the component methods. If you pass the * parameter, it accesses the methods of the host database.

## Example 1

You want to find out modification dates and times for several methods:

```
ARRAY TEXT(arrPaths;0)
APPEND TO ARRAY(arrPaths;"MyMethod1")
APPEND TO ARRAY(arrPaths;"MyMethod2")
...
ARRAY DATE(arrDates;0)
ARRAY LONGINT(arrTimes;0)
METHOD GET MODIFICATION DATE(arrPaths;arrDates;arrTimes)
```

## Example 2

You want to get modification dates for methods in a module that are prefixed with "Web_". You cannot use the "@" symbol in a path; however, you can write:

```
ARRAY TEXT($_webMethod;0)
METHOD GET NAMES($_webMethod;"Web_@")
ARRAY DATE($_date;0)
ARRAY LONGINT($_time;0)
METHOD GET MODIFICATION DATE($_webMethod;$_date;$_time)
```

# METHOD GET NAMES

METHOD GET NAMES ( arrNames {; filter}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrNames | Text array | ⇦ | Array of project method names |
| filter | Text | ⇨ | Name filter |
| * | Operator | ⇨ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD GET NAMES** command fills the *arrNames* array with the names of project methods created in the application.

By default, all methods are listed. You can restrict this list by passing a comparison string in the *filter* parameter: in this case, the command only returns methods whose name matches the filter. You must use the @ character in order to set filters of the "starts with", "ends with" or "contains" type. If you pass an empty string, the *filter* parameter is ignored.

If this command is executed from a component, it returns by default the names of the component project methods. If you pass the * parameter, the array contains the host database project methods.

**Note:** Methods placed in the trash are not listed.

## Example

Here are a few typical examples of use:

```
 // List of all project methods of the database
METHOD GET NAMES(t_Names)

 //  List of project methods beginning with a specific string
METHOD GET NAMES(t_Names;"web_@")

 // List of project methods in the host database beginning with a specific string
METHOD GET NAMES(t_Names;"web_@";*)
```

# METHOD Get path

METHOD Get path ( methodType {; aTable}{; objectName{; formObjectName}}{; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| methodType | Longint | ⇒ | Object type selector |
| aTable | Table | ⇒ | Table reference |
| objectName | Text | ⇒ | Name of form or database method |
| formObjectName | Text | ⇐ | Name of form object |
| * | Operator | ⇒ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |
| Function result | Text | ⇒ | Full path of object |

## Description

The **METHOD Get path** command returns the full internal pathname of a method.

In the *methodType* parameter, you pass the type of method whose path you want to get. You can use the following constants, found in the **Design Object Access** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Path database method | Longint | 2 | Path of database methods specified (English name). List of these methods:<br>*[databaseMethod]/onStartup*<br>*[databaseMethod]/onExit*<br>*[databaseMethod]/onDrop*<br>*[databaseMethod]/onBackupStartup*<br>*[databaseMethod]/onBackupShutdown*<br>*[databaseMethod]/onWebConnection*<br>*[databaseMethod]/onWebAuthentication*<br>*[databaseMethod]/onWebSessionSuspend*<br>*[databaseMethod]/onServerStartup*<br>*[databaseMethod]/onServerShutdown*<br>*[databaseMethod]/onServerOpenConnexion*<br>*[databaseMethod]/onServerCloseConnection*<br>*[databaseMethod]/onSystemEvent*<br>*[databaseMethod]/onSqlAuthentication* |
| Path project form | Longint | 4 | Path of project form methods and all their object methods. Examples:<br>*[projectForm]/myForm/{formMethod}*<br>*[projectForm]/myForm/button1*<br>*[projectForm]/myForm/my%2list*<br>*[projectForm]/myForm/button1* |
| Path project method | Longint | 1 | Name of method.<br>Example: *MyProjectMethod* |
| Path table form | Longint | 16 | Path of table form methods and all their object methods. Example:<br>*[tableForm]/table_1/Form1/{formMethod}*<br>*[tableForm]/table_1/Form1/button1*<br>*[tableForm]/table_1/Form1/my%2list*<br>*[tableForm]/table_2/Form1/my%2list* |
| Path trigger | Longint | 8 | Path of database triggers. Example:<br>*[trigger]/table_1*<br>*[trigger]/table_2* |

Pass values in the *aTable*, *objectName* and *formObjectName* parameters according to the type of object for which you want to get the method pathname:

| **Type of object** | *aTable* | *objectName* | *formObjectName* |
|---|---|---|---|
| Path Project form | | X | X (optional) |
| Path Table form | X | X | X (optional) |
| Path Database method | | X | |
| Path Project method | | X | |
| Path Trigger | X | | |

If the object is not found (method type unknown or not valid, missing table, etc.), an error is generated.

If the command is executed from a component, it returns by default the paths of the component methods. If you pass the * parameter, the array contains the paths of the methods of the host database.

## Example

```
 //Retrieval of the pathname of the "On Startup" database method:
$path:=METHOD Get path(Path database method;"onStartup")

 //Retrieval of the pathname of the trigger for the [Employees] table:
$path:=METHOD Get path(Path trigger;[Employees])

 //Retrieval of the pathname of the "OK" object method of the "input" form for the [Employees] table:
$path:=METHOD Get path(Path table form;[Employees];"input";"OK")
```

# METHOD GET PATHS

METHOD GET PATHS ( {folderName ;} methodType ; arrPaths {; stamp}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| folderName | Text | ⇒ | Name of Home page folder |
| methodType | Longint | ⇒ | Selector of method type to get |
| arrPaths | Text array | ⇐ | Array of method paths and names |
| stamp | Longint variable | ⇒ | Minimum value of stamp |
| | | ⇐ | New current value |
| * | Operator | ⇒ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD GET PATHS** command fills the *arrPaths* array with the internal pathnames and names of methods in the application that are of the type specified by the *methodType* parameter.

If your code is organized in "folders" in the 4D Explorer (Home page), you can pass a folder name in the optional *folderName* parameter. In this case, the *arrPaths* array only contains paths of methods that are found in this location.

**Note:** You cannot use the "@" character in *folderName*.

In the *methodType* parameter, you pass the type of method whose paths you want to get in the *arrPaths* array. You can use the following constants (individually or a combination of them), found in the **Design Object Access** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Path all objects | Longint | 31 | Paths of all the methods of the database |
| Path database method | Longint | 2 | Path of database methods specified (English name). List of these methods:<br>*[databaseMethod]/onStartup*<br>*[databaseMethod]/onExit*<br>*[databaseMethod]/onDrop*<br>*[databaseMethod]/onBackupStartup*<br>*[databaseMethod]/onBackupShutdown*<br>*[databaseMethod]/onWebConnection*<br>*[databaseMethod]/onWebAuthentication*<br>*[databaseMethod]/onWebSessionSuspend*<br>*[databaseMethod]/onServerStartup*<br>*[databaseMethod]/onServerShutdown*<br>*[databaseMethod]/onServerOpenConnexion*<br>*[databaseMethod]/onServerCloseConnection*<br>*[databaseMethod]/onSystemEvent*<br>*[databaseMethod]/onSqlAuthentication* |
| Path project form | Longint | 4 | Path of project form methods and all their object methods. Examples:<br>*[projectForm]/myForm/{formMethod}*<br>*[projectForm]/myForm/button1*<br>*[projectForm]/myForm/my%2list*<br>*[projectForm]/myForm/button1* |
| Path project method | Longint | 1 | Name of method.<br>Example: *MyProjectMethod* |
| Path table form | Longint | 16 | Path of table form methods and all their object methods. Example:<br>*[tableForm]/table_1/Form1/{formMethod}*<br>*[tableForm]/table_1/Form1/button1*<br>*[tableForm]/table_1/Form1/my%2list*<br>*[tableForm]/table_2/Form1/my%2list* |
| Path trigger | Longint | 8 | Path of database triggers. Example:<br>*[trigger]/table_1*<br>*[trigger]/table_2* |

The *stamp* parameter lets you only get the paths of methods modified after a specific point in time. As part of a version control system, this means that you can update only methods that were modified since the last backup.
Here is how it works: 4D maintains a counter of method modifications. Each time a method is created or saved again, this counter is incremented and its current value is stored in the internal stamp of the method.
If you pass the *stamp* parameter, the command only returns methods whose stamp is greater than or equal to the value passed in this parameter. Moreover, the command returns, in *stamp*, the new current value of the modification counter, i.e. the highest value. If you save this value, you can pass it the next time this command is called so that you only get new or modified methods.

If the command is executed from a component, it returns by default the paths of the component methods. If you pass the * parameter, the array contains the paths of the methods of the host database.

If the command detects a duplicated method name, the error -9802 is generated ("Object path not unique"). In this case, it is advisable to use the MSC in order to verify the database structure.

## Example 1

Retrieval of project methods found in the "web" folder:

```
METHOD GET PATHS("web"; Path Project method;arrPaths)
```

## Example 2

Retrieval of database methods and triggers:

```
METHOD GET PATHS(Path trigger+Path database method;arrPaths)
```

## Example 3

Retrieval of project methods that were modified since the last backup:

```
 // we load the last saved value
$stamp :=Max([Backups]cur_stamp)
METHOD GET PATHS(Path project method;arrPaths;$stamp)
 // we save the new value
CREATE RECORD([Backups])
[Backups]cur_stamp :=$stamp
SAVE RECORD([Backups])
```

## Example 4

Refer to the example of the **METHOD SET CODE** command.

# METHOD GET PATHS FORM

METHOD GET PATHS FORM ( {aTable ;} arrPaths {; filter}{; stamp}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table reference |
| arrPaths | Text array | ⬅ | Array of method paths and names |
| filter | Text | ➡ | Name filter |
| stamp | Longint variable | ➡ | Minimum value of stamp |
| | | ⬅ | New current value |
| * | Operator | ➡ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD GET PATHS FORM** command fills the *arrPaths* array with the internal pathnames and names of the methods for all form objects as well as form methods. Form methods are labeled {formMethod}.

Only objects containing code are listed. For example, buttons that are only associated with a standard action are not returned.

If you pass the *aTable* parameter, the command returns the objects of the table forms associated with this table. If you omit this parameter, the command returns objects of the database project forms.

You can limit this list of forms by passing a comparison string in the *filter* parameter: in this case, only forms whose names match the filter are returned. You can use the @ character in order to specify "starts with", "ends with" or "contains" type filters. If you pass an empty string, the *filter* parameter is ignored.

The *stamp* parameter lets you only get the paths of methods modified after a specific point in time. As part of a version control system, this means that you can update only methods that were modified since the last backup.
Here is how it works: 4D maintains a counter of method modifications. Each time a method is created or saved again, this counter is incremented and its current value is stored in the internal stamp of the method.
If you pass the *stamp* parameter, the command only returns methods whose stamp is greater than or equal to the value passed in this parameter. Moreover, the command returns, in *stamp*, the new current value of the modification counter, i.e. the highest value. If you save this value, you can pass it the next time this command is called so that you only get new or modified methods.

If the command is executed from a component, it returns by default the paths of the component methods. If you pass the * parameter, the array contains the paths of the methods of the host database.

**Note:** The command does not list objects of inherited forms or of subforms.

If the command detects a duplicated method name, the error -9802 is generated ("Object path not unique"). In this case, it is advisable to use the MSC in order to verify the database structure.

## Example 1

List of all objects of the "input" form for the [Employees] table. Note that table form methods (and project form methods) are processed as objects belonging to the form:

```
METHOD GET PATHS FORM([Employees];arrPaths;"input")
 // Contents of arrPaths (for example)
 // [tableForm]/input/{formMethod} -> Form method
 // [tableForm]/input/bOK -> Object method
 // [tableForm]/input/bCancel -> Object method
```

## Example 2

List of objects of the "dial" project form:

```
METHOD GET PATHS FORM(arrPaths;"dial")
```

## Example 3

List of all objects of the "input" form for the [Employees] table from a component:

```
METHOD GET PATHS FORM(([Employees];arrPaths;"input@";*)
```

# METHOD OPEN PATH

METHOD OPEN PATH ( path {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text | ⇒ | Path of method to open |
| * | Operator | ⇒ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD OPEN PATH** command opens, in the 4D Method editor, the method whose internal pathname is passed in the *path* parameter.

This command can open all method types (object, form, trigger, project or database); however, the method must already exist. If the *path* parameter does not correspond to an existing method, the error -9801 "Cannot open method" is returned.

You can execute this command from a component, but in this case, you must pass the * parameter because access to the component code is read-only. If you omit the * parameter in this context, the error -9763 is generated

# METHOD RESOLVE PATH

METHOD RESOLVE PATH ( path ; methodType ; ptrTable ; objectName ; formObjectName {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text | ⇒ | Path to resolve |
| methodType | Longint | ⇐ | Object type selector |
| ptrTable | Pointer | ⇐ | Table reference |
| objectName | Text | ⇐ | Name of form or database method |
| formObjectName | Text | ⇐ | Name of form object |
| * | Operator | ⇒ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD RESOLVE PATH** command parses the internal path name passed in the *path* parameter and returns its different components in the *methodType*, *ptrTable*, *objectName*, and *formObjectName* parameters.

The *methodType* parameter receives a value indicating the type of the method. You can compare this value with the following constants of the **Design Object Access** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Path database method | Longint | 2 | Path of database methods specified (English name). List of these methods:<br>*[databaseMethod]/onStartup*<br>*[databaseMethod]/onExit*<br>*[databaseMethod]/onDrop*<br>*[databaseMethod]/onBackupStartup*<br>*[databaseMethod]/onBackupShutdown*<br>*[databaseMethod]/onWebConnection*<br>*[databaseMethod]/onWebAuthentication*<br>*[databaseMethod]/onWebSessionSuspend*<br>*[databaseMethod]/onServerStartup*<br>*[databaseMethod]/onServerShutdown*<br>*[databaseMethod]/onServerOpenConnexion*<br>*[databaseMethod]/onServerCloseConnection*<br>*[databaseMethod]/onSystemEvent*<br>*[databaseMethod]/onSqlAuthentication* |
| Path project form | Longint | 4 | Path of project form methods and all their object methods. Examples:<br>*[projectForm]/myForm/{formMethod}*<br>*[projectForm]/myForm/button1*<br>*[projectForm]/myForm/my%2list*<br>*[projectForm]/myForm/button1* |
| Path project method | Longint | 1 | Name of method.<br>Example: *MyProjectMethod* |
| Path table form | Longint | 16 | Path of table form methods and all their object methods. Example:<br>*[tableForm]/table_1/Form1/{formMethod}*<br>*[tableForm]/table_1/Form1/button1*<br>*[tableForm]/table_1/Form1/my%2list*<br>*[tableForm]/table_2/Form1/my%2list* |
| Path trigger | Longint | 8 | Path of database triggers. Example:<br>*[trigger]/table_1*<br>*[trigger]/table_2* |

The *ptrTable* parameter contains a pointer to a database table when the path references a table form method or a trigger.

The *objectName* parameter contains either:

- A form name when the path references a table form or project form
- A database method name when the path references a database method.

The *formObjectName* parameter contains a form object name when the path references an object method.

If the command is executed from a component, it considers by default that *path* designates a component method. If you pass the * parameter, then it considers that *path* designates a host database method.

## Example 1

Resolution of a database method path:

```
C_LONGINT($methodType)
C_POINTER($tablePtr)
C_TEXT($objectName)
C_TEXT($formObjectName)

METHOD RESOLVE PATH("[databaseMethod]/onStartup";$methodType;$tablePtr;$objectName;$formObjectName)
 // $methodType: 2
 // $tablePtr: Nil pointer
 // $objectName: "onStartup"
 // $formObjectName: ""
```

## Example 2

Resolution of a path for an object of a table form method:

```
C_LONGINT($methodType)
C_POINTER($tablePtr)
C_TEXT($objectName)
C_TEXT($formObjectName)

METHOD RESOLVE PATH("[tableForm]/Table1/output%2A1/myVar%2A1";$methodType;$tablePtr;$objectName;$formObjectName)
 // $methodType: 16
 // $tablePtr: -> [Table1]
 // $objectName: "output*1"
 // $formObjectName: "Btn*1"
```

# METHOD SET ACCESS MODE

METHOD SET ACCESS MODE ( mode )

| Parameter | Type | | Description |
|---|---|---|---|
| mode | Longint | ⇒ | Access mode for locked objects |

## Description

The **METHOD SET ACCESS MODE** command sets the behavior for 4D when you attempt to write access an object already loaded for modification by another user or process. The scope of this command is the current session.

In *mode*, you pass one of the following constants of the **Design Object Access** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| On object locked abort | Longint | 0 | Loading of object is aborted (default functioning) |
| On object locked confirm | Longint | 2 | 4D displays a dialog box so that you can choose to try again or to abort. In remote mode, this option is not supported (loading is aborted) |
| On object locked retry | Longint | 1 | 4D keeps attempting to load the object until it has been released |

# METHOD SET ATTRIBUTE

METHOD SET ATTRIBUTE ( path ; attribType ; attribValue {; attribType2 ; attribValue2 ; ... ; attribTypeN ; attribValueN}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text | ⇒ | Path of project method |
| attribType | Longint | ⇒ | Type of attribute |
| attribValue | Boolean, Text | ⇒ | True = select attribute, False = deselect attribute or Folder name |
| * | Operator | ⇒ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD SET ATTRIBUTE** command sets the value(s) of the *attribType* attribute(s) for the project method designated by the *path* parameter. This command only works with project methods. If you pass an invalid *path*, an error is generated.

In the *attribType* parameter, pass a value indicating the type of attribute to set. You can use the following constants, found in the **Design Object Access** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Attribute executed on server | Longint | 8 | Corresponds to the "Execute on server" option |
| Attribute folder name | Longint | 1024 | Name of folder for the method ("folder" attribute). When you pass this constant, you must pass a folder name in *attribValue*:<br>• if this name corresponds to a valid folder, the method is placed in this parent folder,<br>• if the folder does not exist, the command does not change anything at the parent folder level,<br>• if you pass an empty string, the method is placed at the root level. |
| Attribute invisible | Longint | 1 | Corresponds to the "Invisible" option |
| Attribute published SOAP | Longint | 3 | Corresponds to the "Offered as a Web Service" option |
| Attribute published SQL | Longint | 7 | Corresponds to the "Available through SQL" option |
| Attribute published Web | Longint | 2 | Corresponds to the "Available through 4D HTML tags and URLs (4DACTION...)" option |
| Attribute published WSDL | Longint | 4 | Corresponds to the "Published in WSDL" option |
| Attribute shared | Longint | 5 | Corresponds to the "Shared by components and host database" option |

In the *attribValue* parameter, you can pass either:

- **True** to select the corresponding option or **False** to deselect it, or
- a string (folder name) if you used the Attribute folder name constant in *attribType*.

You can pass multiple *attribType*;*attribValue* pairs in a single call.

You can execute this command from a component, but in this case, you must pass the * parameter because access to the component code is read-only. If you omit the * parameter in this context, the error -9763 is generated.

This command cannot be executed in compiled mode. When it is called in this mode, it will generate the error -9762.

## Example 1

Selection of the "Shared by components and host database" property for the "Choose dialog" project method:

```
METHOD SET ATTRIBUTE("Choose dialog";Attribute shared;True)
```

## Example 2

Setting several attribute/value pairs:

```
METHOD SET ATTRIBUTE(vPath;Attribute invisible;vInvisible;Attribute published Web;v4DAction;Attribute published SOAP;vSoap;Attribute published WSDL;vWSDL;Attribute shared;vExported;Attribute published SQL;vSQL;Attribute executed on server;vRemote;Attribute folder name;vFolder;*)
```

# METHOD SET ATTRIBUTES

METHOD SET ATTRIBUTES ( path ; attributes {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text, Text array | ⇒ | Method path(s) |
| attributes | Object, Object array | ⇒ | Attribute(s) to set for selected method(s) |
| * | Operator | ⇒ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD SET ATTRIBUTES** command allows you to set the *attributes* values for the method(s) specified in the *path* parameter.

In *path*, you can pass either a text containing a method path, or a text array containing an array of paths. You will need to pass the same kind of parameter (string or array) in *attributes* in order to set the appropriate attributes. This command only works with project methods. If you pass an invalid *path*, an error is generated.

In *attributes*, you pass an object or an array of objects (depending on the kind of parameter you passed in *path*) containing all the attributes that you want to set for the method(s).

Method attributes must be set using the **OB SET** or **OB SET ARRAY** commands, with True or False values for Boolean attributes, or specific values for extended attributes (for example, "scope":"table" for the 4D Mobile property). Only attributes that are present in the *attributes* parameter will be updated in the method attributes.

If the command is executed from a component, by default it applies to the component methods. If you pass the * parameter, it accesses the methods of the host database.

**Note:** The existing **METHOD SET ATTRIBUTE** command is still supported but since it can only handle Boolean values, it cannot be used for extended attributes such as 4D Mobile properties.

The supported attributes are:

```
{    "invisible" : false, // true if visible    "preemptive" : "capable" // or "incapable" or "indifferent"    "publishedWeb" : false,  // true if available through 4D tags and URLs    "publishedSoap": false,  // true if offered as Web Service    "publishedWsdl": false,  // true if published in WSDL    "shared" : false,  // true if shared by components and host database    "publishedSql" : false,  // true if available through SQL "executedOnServer" : false, // true if executed on server    "published4DMobile" : {        "scope": "table",  // "none" or "table" or "currentRecord" or "currentSelection"        "table": "aTableName"  // present if scope is different from "none"    } }
```

**Note:** Regarding "published4DMobile" attributes, if the "table" value does not exist or if the "scope" is invalid, these attributes are ignored.

## Example 1

You want to set a single attribute:

```
C_OBJECT($attributes)
OB SET($attributes;"executedOnServer";True)
METHOD SET ATTRIBUTES("aMethod";$attributes) //Only the "executedOnServer" attribute is modified
```

## Example 2

You want a method to be unavailable through 4D Mobile (the "none" value must be passed for the "scope" attribute):

```
C_OBJECT($attributes)
C_OBJECT($fourDMobileAttribute)
OB SET($fourDMobileAttribute;"scope";"none")
OB SET($attributes;"published4DMobile";$fourDMobileAttribute)
METHOD SET ATTRIBUTES("aMethod";$attributes)
```

# METHOD SET CODE

METHOD SET CODE ( path ; code {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text, Text array | ➡ | Text or Text array containing one or more method path(s) |
| code | Text, Text array | ➡ | Code of designated method(s) |
| * | Operator | ➡ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD SET CODE** command modifies the code of the method(s) designated by the *path* parameter with the contents passed in the *code* parameter. This command can access the code of all types of methods: database methods, triggers, project methods, form methods and object methods.

In the case of a project method: if this method already exists in the database, its contents are replaced; if it does not exist, it is created with its contents.

You can use two types of syntaxes, based either on text arrays, or text variables:

```
C_TEXT(tVpath) // text variables
C_TEXT(tVcode)
METHOD SET CODE(tVpath;tVcode) // code of a single method
```

```
ARRAY TEXT(arrPaths;0) // text arrays
ARRAY TEXT(arrCodes;0)
METHOD SET CODE(arrPaths;arrCodes) // code of several methods
```

You cannot mix the two syntaxes.

If you pass an invalid pathname, the command does nothing. .

When **METHOD SET CODE** is called, the method attributes are reset by default. However, if the first line of the method *code* contains valid metadata (expressed in JSON), they are used to specify the method attributes and the first line is not inserted. Example of metadata:

```
// %attributes = {"invisible":true,"lang":"fr","folder":"Security"}
```

**Note:** These metadata are generated automatically by the **METHOD GET CODE** command. For more information about supported attributes, refer to the description of the **METHOD SET ATTRIBUTES** command.

Concerning the "folder" property of the metadata:

- When this property is present and corresponds to a valid folder, the method is placed in its parent folder,
- If this property is not present or if the folder does not exist, the command makes no changes at the parent folder level,
- When this property contains an empty string, the method is placed at the root level.

You can execute this command from a component, but in this case, you must pass the * parameter because access to the component code is read-only. If you omit the * parameter in this context, the error -9763 is generated

## Example

This example exports and imports all the project methods of an application:

```
$root_t:=Get 4D folder(Database folder)+"methods"+Folder separator
ARRAY TEXT($fileNames_at;0)
CONFIRM("Import or export methods?";"Import";"Export")

If(OK=1)
   DOCUMENT LIST($root_t;$fileNames_at)
```

```
      For($loop_l;1;Size of array($fileNames_at))
         $filename_t:=$fileNames_at{$loop_l}
         DOCUMENT TO BLOB($root_t+$filename_t;$blob_x)
         METHOD SET CODE($filename_t;BLOB to text($blob_x;UTF8 text without length))
      End for
   Else
      If(Test path name($root_t)#Is a folder)
         CREATE FOLDER($root_t;*)
      End if
      METHOD GET PATHS(Path project method;$fileNames_at)
      METHOD GET CODE($fileNames_at;$code_at)
      For($loop_l;1;Size of array($fileNames_at))
         $filename_t:=$fileNames_at{$loop_l}
         SET BLOB SIZE($blob_x;0)
         TEXT TO BLOB($code_at{$loop_l};$blob_x;UTF8 text without length)
         BLOB TO DOCUMENT($root_t+$filename_t;$blob_x)
      End for
   End if
   SHOW ON DISK($root_t)
```

# METHOD SET COMMENTS

METHOD SET COMMENTS ( path ; comments {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | Text, Text array | ➡ | Text or Text array containing one or more method path(s) |
| comments | Text, Text array | ➡ | Comments of designated method(s) |
| * | Operator | ➡ | If passed = command applies to host database when executed from a component (parameter ignored outside of this context) |

## Description

The **METHOD SET COMMENTS** command replaces the comments of the method(s) designated by the *path* parameter by those specified in the *comments* parameter.

The comments modified by this command are those specified in the 4D Explorer (not to be confused with lines of comments in the code). These comments can only be generated for triggers, project methods and form methods. They contain styled text

**Note:** Forms and form methods share the same comments..

You can use two types of syntaxes, based either on text arrays, or text variables:

```
C_TEXT(tVpath) // text variables
C_TEXT(tVcomments)
METHOD SET COMMENTS(tVpath;tVcomments) // comments for a single method
```

```
ARRAY TEXT(arrPaths;0) // text arrays
ARRAY TEXT(arrComments;0)
METHOD SET COMMENTS(arrPaths;arrComments) // comments for several methods
```

You cannot mix the two syntaxes.

If you pass an invalid pathname, an error is generated.

You can execute this command from a component, but in this case, you must pass the * parameter because access to the component code is read-only. If you omit the * parameter in this context, the error -9763 is generated

## Example

Add a modification date to an existing trigger comment:

```
METHOD GET COMMENTS("[trigger]/Table1";$comments)
$comments:="Modif:"+String(Current date)+"¥r"+$comments
METHOD SET COMMENTS("[trigger]/Table1";$comments)
```

# Drag and Drop

- Drag and Drop
- On Drop Database Method
- DRAG AND DROP PROPERTIES
- Drop position
- SET DRAG ICON

# Drag and Drop

4D allows built-in drag and drop capability between objects in your forms and applications. You can drag and drop one object to another, in the same window or in another window. In other words, drag and drop can be performed within a process or from one process to another.

You can also drag and drop objects between 4D forms and other applications, and vice versa. For example, it is possible to drag and drop a GIF picture file onto a 4D picture field. It is also possible to select text in a word processing application and drop it onto a 4D text variable.

Finally, it is possible to drop objects directly onto the application without necessarily having a form in the foreground. The **On Drop Database Method** can be used to manage the drag and drop action in this case. This means, for example, that you can open a 4D Write document by dropping it onto the 4D application icon.

**Note**: As an introduction, we assume that a drag and drop action “transports” some data from one point to another. Later, we will see that drag and drop can also be a metaphor for any type of operation.

## Draggable and Droppable Object Properties

To drag and drop an object to another object, you must select the **Draggable property** for that object in the Property List window. In a drag-and-drop operation, the object that you drag is the **source object**.

To make an object the destination of a drag and drop operation, you must select the **Droppable property** for that object in the Property List window. In a drag-and-drop operation, the object that receives data is the **destination object**.

**Automatic Drag** and **Automatic Drop**: These additional properties are available for text fields and variables as well as for combo boxes and list boxes. The Automatic Drop option is also available for picture fields and variables. They can be used to enable an automatic drag and drop mode based on copying the contents (the drag and drop action is no longer managed by 4D form events). Please refer to the "Automatic Drag and Drop" paragraph at the end of this section.

By default, newly created objects can be neither dragged nor dropped. It is up to you to set these properties.

All objects in an input or dialog form can be made to be dragged and dropped. Individual elements of an array (i.e., scrollable area), items of a hierarchical list or rows in a list box can be dragged and dropped. Conversely, you can drag and drop an object onto an individual element of an array or an item of a hierarchical list or a list box row. However, you cannot drag and drop objects from the detail area of an output form.

You can also manage dragging and dropping onto the application, outside of any form, using the **On Drop Database Method**.

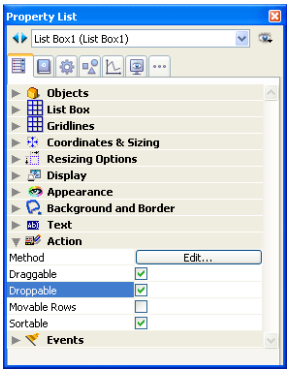
You can easily create a drag-and-drop user interface, because 4D allows you to use any type of active object (field or variable) as source or destination objects. For example, you can drag and drop a button.

**Notes**:

- To drag a text or a button labeled "draggable," you must first press the **Alt** (Windows) or **Option** (Mac OS) keys.
- By default, in the case of picture fields and variables, the picture and its reference are both dragged. If you only want to drag the reference of the variable or field, first hold down the **Alt** (Windows) or **Option** (Mac OS) key.
- When the “Draggable” and “Movable Rows” properties are both set for a List box object, the “Movable Rows” property takes priority when a row is moved. Dragging is not possible in this case.

An object that is capable of being both dragged and dropped can also be dropped onto itself, unless you reject the operation. For details, see the discussion below.

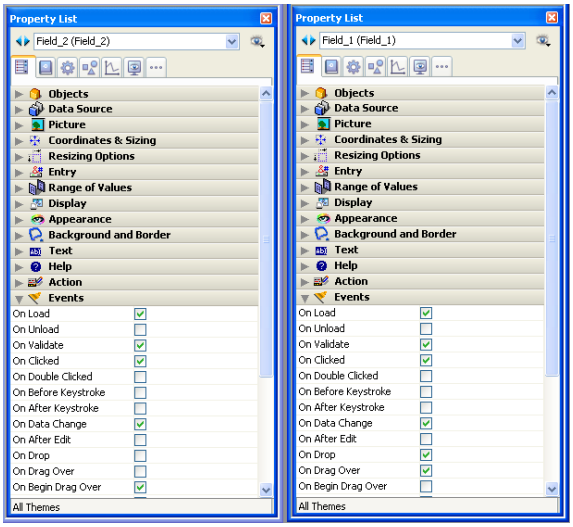
The following figure shows the Property List window with the Droppable and Draggable properties set for the selected object:




## Drag-and-Drop Programmatical Handling

Management of drag and drop by programming is based on three form events: On Begin Drag Over, On Drag Over and On Drop.
Note that the On Begin Drag Over event **is generated in the context of the source object** of the drag while On Drag Over and On Drop **are only sent to the destination object**.

In order for the application to process these events, they must be selected in an appropriate manner in the Property List:
**Property List**:




### On Begin Drag Over

The On Begin Drag Over form event can be selected for any form objects that can be dragged. It is generated in every case where the object has the **Draggable** property.
Unlike the On Drag Over form event, On Begin Drag Over is called within the context of the source object of the drag action. It can be called from the method of the source object or the form method of the source object.
This event is useful for advanced management of the drag action. It can be used to:

- Get the data and signatures found in the pasteboard (via the **GET PASTEBOARD DATA** command).
- Add data and signatures to the pasteboard (via the **APPEND DATA TO PASTEBOARD** command).
- Use a custom icon during the drag action (via the **SET DRAG ICON** command).
- Accept or refuse dragging via $0 in the method of the dragged object. To indicate that drag actions are accepted, the method of the source object must return *0* (zero); you must therefore execute *$0:=0*. To indicate that drag actions are refused, the method of the source object must return *-1* (minus one); you must therefore execute *$0:=-1*. If no result is returned, 4D considers that drag actions are accepted.

4D data are put in the pasteboard before calling the event. For example, in the case of dragging without the **Automatic Drag** action, the dragged text is already in the pasteboard when the event is called.

### On Drag Over

The On Drag Over event is repeatedly sent to the destination object when the mouse pointer is moved over the object. In response to this event, you usually:

- Call the **DRAG AND DROP PROPERTIES** command, which informs you about the source object.
- Depending on the nature and type of both the destination object (whose object method is currently being executed) and the source object, you **accept** or **reject** the drag and drop.

To accept the drag, the destination object method must return *0* (zero), so you write *$0:=0*. To reject the drag, the object method must return *-1* (minus one), so you write *$0:=-1*. During an On Drag Over event, 4D treats the object method as a function. If no result is returned, 4D assumes that the drag is accepted.

If you accept the drag, the destination object is highlighted. If you reject the drag, the destination is not highlighted. Accepting the drag does not mean that the dragged data is going to be inserted into the destination object. It only means that if the mouse button was released at this point, the destination object would accept the dragged data.

If you do not process the On Drag Over event for a droppable object, that object will be highlighted for all drag over operations, no matter what the nature and type of the dragged data.

The On Drag Over event is the means by which you control the first phase of a drag-and-drop operation. Not only can you test whether the dragged data is of a type compatible with the destination object, and then accept or reject the drag; you can simultaneously notify the user of this fact, because 4D highlights (or not) the destination object, based on your decision.

The code handling an On Drag Over event should be short and execute quickly, because that event is sent repeatedly to the current destination object, due to the movements of the mouse.

**WARNING**: Beginning with version 11 of 4D, if the drag and drop is an **interprocess drag and drop**, which means the source object is located in a process (window) other than that of the destination object, the object method of the destination object for an On Drag Over event is executed **within the context of the destination process**. To find out the value of the elements being dragged, you must use the interprocess communication commands. It is usually recommended in this case to use the On Begin Drag Over event and the commands of the **Pasteboard** theme.

### On Drop

The On Drop event is sent once to the destination object when the mouse pointer is released over the object. This event is the second phase of the drag-and-drop operation, in which you perform an operation in response to the user action.

This event is not sent to the object if the drag was not accepted during the On Drag Over events. If you process the On Drag Over event for an object and reject a drag, the On Drop event does not occur. Thus, if during the On Drag Over event you have tested the data type compatibility between the source and destination objects and have accepted a possible drop, you do not need to re-test the data during the On Drop. You already know that the data is suitable for the destination object.

An interesting aspect of the 4D drag-and-drop implementation is that 4D lets you do whatever you want. Examples:

- If a hierarchical list item is dropped over a text field, you can insert the text of the list item at the beginning, at the end, or in the middle of the text field.
- Your form contains a two-state picture button, which could represent an empty or full trash can. Dropping an object onto that button could mean (from the user interface standpoint) “delete the object that has been dragged and dropped into the trash can.” Here, the drag and drop does not transport data from one point to another; instead, it performs an action.
- Dragging an array element from a floating window to an object in a form could mean “in this window, show the Customer record whose name you just dragged and dropped from the floating window listing the Customers stored in the database.”
- And so on.

So, the 4D drag-and-drop interface is a framework which enables you to implement any user interface metaphor you may devise.

## Drag-and-drop commands

The **DRAG AND DROP PROPERTIES** command returns:

- A pointer to the dragged object (field or variable)
- The element or item number, if the dragged object is an array element or a list item
- The process number of the source process.

The **Drop position** command returns the element number of the item position of the target element or list item, if the destination object is an array (i.e., scrollable area), a hierarchical list, a text or a combo box, as well as the column number if the object is a list box.

Commands like **RESOLVE POINTER** and **Type** are useful for testing the nature and type of the source object.

When the drag-and-drop operation is intended to copy the dragged data, the functionality of these commands depend on how many processes are involved:

- If the drag and drop is limited to one process, use these commands to perform the appropriate actions (i.e., simply assigning the source object to the destination object).
- If the drag and drop is an interprocess drag and drop, you need to be careful while getting access to the dragged data; you must access the data instance from the source process. If the dragged data comes from a variable, use **GET**

**PROCESS VARIABLE** to get the right value. If the dragged data comes from a field, remember that the current record for a table is probably different for the two processes, so you need to access the right record. It is usually recommended in this case to use the commands of the **Pasteboard** theme and the On Begin Drag Over event.

If the drag and drop is not intended to move data, but is instead a user interface metaphor for a particular operation, you can perform whatever you want.

## Commands of the Pasteboard Theme

If the drag and drop operation involves the moving of heterogenous data or documents between two 4D applications or a 4D application and a third-party application, the commands of the "Pasteboard" theme will provide you with the tools needed. In fact, these commands can be used to manage both the copy/paste and the drag and drop of data. 4D uses two pasteboards: one for copied (or cut) data, which is the actual clipboard, and the other for data being dragged and dropped. These two pastebaoards are managed using the same commands. You access one or the other depending on the context.

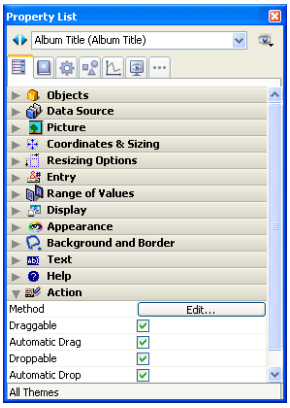
For more information about using the commands of the "Pasteboard" theme for drag and drop operations, please refer to the **Managing Pasteboards** section.

## Automatic Drag and Drop

Text areas (fields, variables, combo boxes and List boxes) as well as picture objects allow the automatic drag and drop, which is the movement or copy of a text or picture selection from one area to another by a single click. It can be used in the same 4D area, between two 4D areas, or between 4D and another application, for example WordPad.

**Note**: In the case of automatic drag and drop between two 4D areas, the data are moved, in other words, they are removed from the source area. If you want to copy the data, hold down the **Ctrl** (Windows) or **Option** (OS X) key during the action (under OS X, you need to hit the **Option** key *after* you start to drag the item(s)).

Automatic drag and drop can be configured separately for each form object via two options of the Property List: **Automatic Drag** and **Automatic Drop**:




- **Automatic Drag** (Text type objects only): When this option is checked, the automatic drag mode is activated for the object. In this mode, the **On Begin Drag** form event is NOT generated.
  If you want to "force" the use of the standard drag while automatic drag is enabled, hold down the **Alt** (Windows) or **Option** (Mac OS) key during the action (under OS X, you need to hit the **Option** key *before* you start to drag the item(s)). This option is not available for pictures.

- **Automatic Drop**: This option is used to activate the automatic drop mode. In this mode, 4D automatically manages — if possible — the insertion of dragged data of the text or picture type that is dropped onto the object (the data are pasted into the object). The On Drag Over and On Drop form events are not generated in this case. On the other hand, the On After Edit (during a drop) and On Data Change (when the object loses the focus) events are generated.
  In the case of data other than text or pictures (another 4D object, file, etc. ) or complex data being dropped, the application refers to the value of the Droppable option: if it is checked, the On Drag Over and On Drop form events are generated; otherwise, the drop is refused. This also depends on the value of the "Prevent drop of data not coming from 4D" option (see below).

## Prevent drop of data not coming from 4D (compatibility)

Beginning with version 11, 4D allows drag and drop of selections, objects and/or files external to 4D, like picture files for example. This possibility must be supported by the database code.
In databases converted from previous versions of 4D, this possibility may lead to malfunctioning if the existing code is not adapted accordingly. For this reason, an option in the Preferences can be used to disable this function: **Prevent drop of**

**data not coming from 4D**. This option is found on the Application/Compatibility page. It is checked by default in converted databases.

When this option is check, the drop of external objects into 4D forms is refused. Note however that the insertion of external objects remains possible in objects having the **Automatic Drop** option, when the application can interpret the dropped data (text or picture).

# On Drop Database Method

The **On Drop Database Method** is available in local or remote 4D applications.

This database method is automatically executed in the case of objects being dropped in the 4D application outside of any form or windows, i.e.:

- In an empty area of the MDI window (Windows),
- On the 4D icon in the Dock (Mac OS) or on the system desktop.

Under Mac OS, you need to hold down the **Option**+**Command** keys during the drop in order for the database method to be called.

When a drop occurs on the 4D application icon on the desktop, the **On Drop Database Method** is only called when the application is already launched, except in the case of applications merged with 4D Desktop. In this case, the database method is called even when the application is not launched. This means that it is possible to define custom document signatures.

## Example

This example can be used to open a 4D Write document that is dropped outside of any form:

```
  `On Drop database method
droppedFile:=Get file from pasteboard(1)
If(Position(".4W7";droppedFile)=Length(droppedFile)-3)
   externalArea:=Open external window(100;100;500;500;0;droppedFile;"_4D Write")
   WR OPEN DOCUMENT(externalArea;droppedFile)
End if
```

# DRAG AND DROP PROPERTIES

DRAG AND DROP PROPERTIES ( srcObject ; srcElement ; srcProcess )

| Parameter | Type | | Description |
|---|---|---|---|
| srcObject | Pointer | ⇐ | Pointer to drag-and-drop source object |
| srcElement | Longint | ⇐ | Dragged array element number, or Dragged list box row number, or Dragged hierarchical list item, or -1 if source object is neither an array nor a list box nor a hierarchical list |
| srcProcess | Longint | ⇐ | Source process number |

## Description

**Compatibility note:** Since version 11 of 4D, it is recommended to manage drag and drop operations, especially interprocess ones, using the On Begin Drag Over event and the commands of the **Pasteboard** theme.

The **DRAG AND DROP PROPERTIES** command enables you to obtain information about the source object when an On Drag Over or On Drop event occurs for a “complex” object (array, list box or hierarchical list).

Typically, you use **DRAG AND DROP PROPERTIES** from within the object method of the object (or from one of the subroutines it calls) for which the On Drag Over or On Drop event occurs (the destination object).

**Important:** A form object accepts dropped data if its **Droppable** property has been selected. Also, its object method must be activated for On Drag Over and/or On Drop, in order to process these events.

After the call:

- The *srcObject* parameter is a pointer to the source object (the object that has been dragged and dropped). Note that this object can be the destination object (the object for which the On Drag Over or On Drop event occurs) or a different object. Dragging and dropping data from and to the same object is useful for arrays and hierarchical lists—it is a simple way of allowing the user to sort an array or a list manually.
- If the dragged and dropped data is an array element (the source object being an array), the *srcElement* parameter returns the number of this element. If the dragged and dropped data is a list box row, the *srcElement* parameter returns the number of this row. If the drag and dropped data is a list item (the source object being a hierarchical list), the *srcElement* parameter returns the position of this item. Otherwise, if the source object does not belong to any of these categories, *srcElement* is equal to -1.
- Drag and drop operations can occur between processes. The *srcProcess* parameter is equal to the number process to which the source object belongs. It is important to test the value of this parameter. You can respond to a drag and drop within the same process by simply copying the source data to the destination object. On the other hand, when treating an interprocess drag and drop, you will use the **GET PROCESS VARIABLE** command to get the source data from the source process object instance. You will usually implement drag and drop in the user interface from source variables (i.e., arrays and lists) toward data entry areas (fields or variables).

If you call **DRAG AND DROP PROPERTIES** when there is no drag and drop event, *srcObject* returns a NIL pointer, *srcElement* returns -1 and *srcProcess* returns 0.

**Tip:** 4D automatically handles the graphical aspect of a drag and drop. You must then respond to the event in the appropriate way. In the following examples, the response is to copy the data that has been dragged. Alternatively, you can implement sophisticated user interfaces where, for example, dragging and dropping an array element from a floating window will fill in the destination window (the window where the destination object is located) with structured data (i.e., several fields coming from a record uniquely identified by the source array element).

You use **DRAG AND DROP PROPERTIES** during an On Drag Over event in order to decide whether the destination object accepts the drag and drop operation, depending on the type and/or the nature of the source object (or any other reason). If you accept the drag and drop, the object method must return *$0:=0*. If you do not accept the drag and drop, the object method must return *$0:=-1*. Accepting or refusing the drag and drop is reflected on the screen—the object is or is not highlighted as the potential destination of the drag-and-drop operation.

## Example 1

In several of your database forms, there are scrollable areas in which you want to manually reorder the elements by simple drag and drop from one part of the scrollable area into another within it. Rather than writing specific code for each case, you may implement a generic project method that will handle any one of these scrollable areas. You could write something like:

```
   ` Handle self array drag and drop project method
   ` Handle self array drag and drop ( Pointer ) -> Boolean
   ` Handle self array drag and drop ( -> Array ) -> Is a self array drag and drop

Case of
   :(Form event=On Drag Over)
      DRAG AND DROP PROPERTIES($vpSrcObj;$vlSrcElem;$vlPID)
      If($vpSrcObj=$1)
   ` Accept the drag and drop if it is from the array to itself
         $0:=0
      Else
         $0:=-1
      End if
   :(Form event=On Drop)
   ` Get the information about the drag and drop source object
      DRAG AND DROP PROPERTIES($vpSrcObj;$vlSrcElem;$vlPID)
   ` Get the destination element number
      $vlDstElem:=Drop position
   ` If the element was not dropped over itself
      If($vlDstElem #$vlSrcElem)
   ` Save dragged element in element 0 of the array
         $1->{0}:=$1->{$vlSrcElem}
   ` Delete the dragged element
         DELETE FROM ARRAY($1->;$vlSrcElem)
   ` If the destination element was beyond the dragged element
         If($vlDstElem>$vlSrcElem)
   ` Decrement the destination element number
            $vlDstElem:=$vlDstElem-1
         End if
   ` If the drag and drop occurred beyond the last element
         If($vlDstElem=-1)
   ` Set the destination element number to a new element at the end of the array
            $vlDstElem:=Size of array($1->)+1
         End if
   ` Insert this new element
         INSERT IN ARRAY($1->;$vlDstElem)
   ` Set its value which was previously saved in the element zero of the array
         $1->{$vlDstElem}:=$1->{0}
   ` The element becomes the new selected element of the array
         $1->:=$vlDstElem
      End if
End case
```

Once you have implemented this project method, you can use it in the following way:

```
   ` anArray Scrollable Area Object Method

Case of
   ` ...
   :(Form event=On Drag Over)
      $0:=Handle self array drag and drop(Self)
   :(Form event=On Drop)
      Handle self array drag and drop(Self)
   ` ...
End case
```

## Example 2

In several of your database forms, you have text enterable areas in which you want to drag and drop data from various sources. Rather than writing specific code for each case, you may implement a generic project method that will handle any one of these text enterable areas. You could write something like:

```
   ` Handle dropping to text area project method
   ` Handle dropping to text area ( Pointer )
   ` Handle dropping to text area ( -> Text or String variable )
```

```
Case of
  ` Use this event for accepting or rejecting the drag and drop
    :(Form event=On Drag Over)
  ` Initialize $0 for rejecting
      $0:=-1
  ` Get the information about the drag and drop source object
      DRAG AND DROP PROPERTIES($vpSrcObj;$vlSrcElem;$vlPID)
  ` In this example, we do not allow drag and drop from an object to itself
      If($vpSrcObj #$1)
  ` Get the type of the data which is being dragged
         $vlSrcType:=Type($vpSrcObj->)
         Case of
            :($vlSrcType=Is text)
  ` OK for text variables
               $0:=0
            :($vlSrcType=Is string var)
  ` String Variable is OK
               $0:=0
            :(($vlSrcType=String array)|($vlSrcType=Text array))
  ` String and Text Arrays are OK
               $0:=0
            :(($vlSrcType=Is longint)|($vlSrcType=Is real)
               If(Is a list($vpSrcObj->))
  ` Hierarchical list is OK
                  $0:=0
               End if
         End case
      End if

  ` Use this event for performing the actual drag and drop action
    :(Form event=On Drop)
      $vtDraggedData:=""
  ` Get the information about the drag and drop source object
      DRAG AND DROP PROPERTIES($vpSrcObj;$vlSrcElem;$vlPID)
  ` Get the type of the variable which has been dragged
      $vlSrcType:=Type($vpSrcObj->)
      Case of
  ` If it is an array
         :(($vlSrcType=String array)|($vlSrcType=Text array))
            If($vlPID #Current process)
  ` Read the element from the source process instance of the variable
               GET PROCESS VARIABLE($vlPID;$vpSrcObj->{$vlSrcElem};$vtDraggedData)
            Else
  ` Copy the array element
               $vtDraggedData:=$vpSrcObj->{$vlSrcElem}
            End if
  ` If it is a list
         :(($vlSrcType=Is real)|($vlSrcType=Is longint))
  ` If it is a list from another process
            If($vlPID #Current process)
  `Get the List Reference from the other process
               GET PROCESS VARIABLE($vlPID;$vpSrcObj->;$vlList)
            Else
               $vlList:=$vpSrcObj->
            End if
  ` If the list exists
            If(Is a list($vpSrcObj->))
  `Get the text of the item whose position was obtained
               GET LIST ITEM($vlList;$vlSrcElem;$vlItemRef;$vsItemText)
               $vtDraggedData:=$vsItemText
            End if
         Else
  ` It is a string or a text variable
            If($vlPID #Current process)
               GET PROCESS VARIABLE($vlPID;$vpSrcObj->;$vtDraggedData)
            Else
               $vtDraggedData:=$vpSrcObj->
            End if
      End case
  ` If there is actually something to drop (the source object may be empty)
```

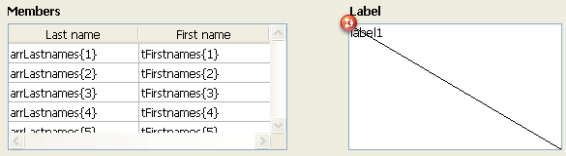
```
      If($vtDraggedData #"")
         $1->:=$1->+$vtDraggedData
      End if
   End case
```

Once you have implemented this project method, you can use it in the following way:

```
   ` [anyTable]aTextField Object Method

   Case of
      ` ...
      :(Form event=On Drag Over)
         $0:=Handle dropping to text area(Self)

      :(Form event=On Drop)
         Handle dropping to text area(Self)
      ` ...
   End case
```

## Example 3

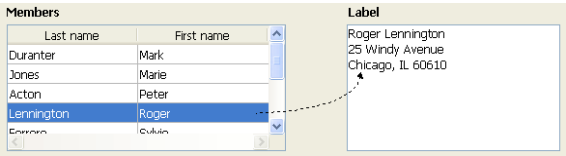
We want to fill a text area (for example, a label) with data dragged from a list box.




Here is the label1 object method:

```
   Case of
      :(Form event=On Drag Over)
         DRAG AND DROP PROPERTIES($source;$arrayrow;$processnum)
         If($source=Get pointer("list box1"))
            $0:=0 `The drop is accepted
         Else
            $0:=-1 `The drag is refused
         End if
      :(Form event=On Drop)
         DRAG AND DROP PROPERTIES($source;$arrayrow;$processnum)
         QUERY([Members];[Members]LastName=arrNames{$arrayrow})
         If(Records in selection([Members])#0)
            label1:=[Members]FirstName+" "+[Members]LastName+Char(Carriage return)+[Members]Address
            +Char(Carriage return)+[Members]City+","+" "+[Members]State
            +" "+[Members]ZipCode
         End if
   End case
```

It then becomes possible to carry out the following action:



# Drop position

Drop position {( columnNumber | pictPosY )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| columnNumber \| pictPosY | Longint | ⇦ | List box column number (-1 if the drop occurs beyond the last column) or<br>Position of Y coordinate in picture |
| Function result | Longint | ⇨ | • Number (array/list box) or<br>• Position (hierarchical list) or<br>• Position in string (text/combo box) of destination item or<br>• -1 if drop occurred beyond the last array element or list item<br>• Position of X coordinate in picture |

## Description

The **Drop position** command can be used to find out the location, in a “complex” destination object, where an object has been (dragged and) dropped.

Typically, you will use **Drop position** when handling a drag and drop event that occurred over an array, a list box, a hierarchical list or a text or picture field.

- If the destination object is an array, the command returns an element number.
- If the destination object is a list box, the command returns a row number. In this case, the command also returns the column number where the drop took place in the optional *columnNumber* parameter.
- If the destination object is a hierarchical list, the command returns an item position.
- If the destination object is a text type variable or field, or a combo box, the command returns a character position within the string.
  In all the above cases, the command may return -1 if the source object has been dropped beyond the last element or the last item of the destination object.
- If the destination object is a picture type variable or field, the command returns the horizontal location of the click and, in the optional *pictPosY* parameter, the vertical location of the click. The values returned are expressed in pixels and in relation to the local coordinate system.

If you call **Drop position** when handling an event that is not a drag-and-drop event and that occurred over an array a list box, a combo box, a hierarchical list, a text or a picture, the command returns -1.

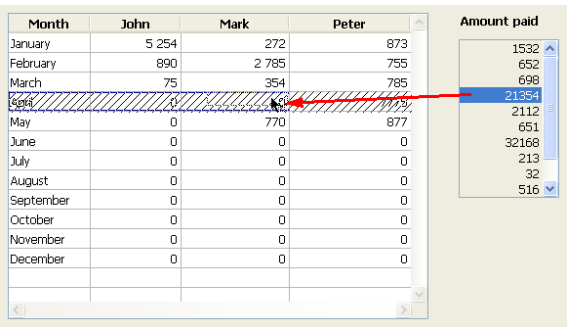
**Important:** A form object accepts dropped data if its **Droppable** property has been selected. Also, its object method must be activated for On Drag Over and/or On Drop, in order to process these events.

## Example 1

See the examples for the **DRAG AND DROP PROPERTIES** command.

## Example 2

In the following example, a list of amounts paid must be broken down per month and per person. This is carried out by drag and drop from a scrollable area:




The list box object method contains the following code:

```
Case of
   :(Form event=On Drag Over)
      DRAG AND DROP PROPERTIES($source;$arrayrow;$processnum)
      If($source=Get pointer("SA1")) `If the drop does come from the scrollable area
         $0:=0
      Else
         $0:=-1 `The drop is refused
      End if
   :(Form event=On Drop)
      DRAG AND DROP PROPERTIES($source;$arrayrow;$processnum)
      $rownum:=Drop position($colnum)
      If($colnum=1)
         BEEP
      Else
         Case of `Adding of dropped values
            :($colnum=2)
               John{$rownum}:=John{$rownum}+SA1{$arrayrow}
            :($colnum=3)
               Mark{$rownum}:=Mark{$rownum}+SA1{$arrayrow}
            :($colnum=4)
               Peter{$rownum}:=Peter{$rownum}+SA1{$arrayrow}
         End case
         DELETE FROM ARRAY(SA1;$arrayrow) `Updating of area
      End if
End case
```

# SET DRAG ICON

SET DRAG ICON ( icon {; horOffset {; vertOffset}} )

| Parameter | Type | | Description |
|---|---|---|---|
| icon | Picture | ⇒ | Icon to use during drag |
| horOffset | Longint | ⇒ | Horizontal offset from left edge of picture with respect to cursor position (>0 = to the left, <0 = to the right) |
| vertOffset | Longint | ⇒ | Vertical offset from top edge of picture with respect to cursor position (>0 = upwards, <0 = downwards) |

## Description

The **SET DRAG ICON** command associates the icon picture with the cursor during drag and drop operations that are managed by programming.

This command can only be called in the context of the On Begin Drag Over form event (see the **Form event** command).

In the *icon* parameter, pass the picture to use. Its maximum size is 256x256 pixels. If one of its dimensions exceeds 256 pixels, it is automatically resized.

In *horOffset* and *vertOffset*, you can pass offset values in pixels:

- for *horOffset*, you pass the horizontal offset from the left edge of the icon with respect to the cursor position. Pass a positive value to apply this offset towards the left or a negative value to apply it towards the right.
- for *vertOffset*, you pass the vertical offset from the top edge of the icon with respect to the cursor position. Pass a positive value to apply this offset upwards or a negative value to apply it downwards.

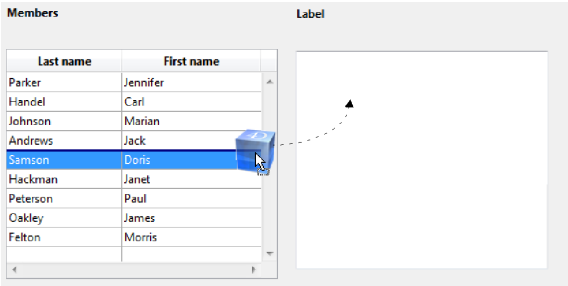
If you omit this parameter, the cursor is placed at the center of the icon.

## Example

In a form, a user can generate a label by dragging and dropping a row. In the object method of the list box, you can write:

```
If(Form event=On Begin Drag Over)
   READ PICTURE FILE(Get 4D folder(Current resources folder)+"splash.png";vpict)
   CREATE THUMBNAIL(vpict;vpict;48;48)
   SET DRAG ICON(vpict)
End if
```

When you drag a row, the picture appears as shown here:




Note that you can modify the position of the cursor with respect to the picture:

```
SET DRAG ICON(vpict;0;0)
```

# Entry Control

- EDIT ITEM
- FILTER KEYSTROKE
- Get edited text
- GET HIGHLIGHT
- GOTO OBJECT
- HIGHLIGHT TEXT
- Keystroke

# EDIT ITEM

EDIT ITEM ( {* ;} object {; item} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If set, object is an object name (string) If omitted, object is a table or variable |
| object | Form object | ⇒ | Object name (if * set) or Table or variable (if * omitted) |
| item | Longint | ⇒ | Item number |

## Description

The **EDIT ITEM** command allows you to edit the current item or the item number *item* in the array or the list set in the *object* parameter.
This means that the selected item can be modified; entering a character entirely replaces the item content.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (in this case, pass a string in *object*). If you do not pass the parameter, you indicate that the *object* parameter is a table or a variable. In this case, you do not pass a string but a table or a variable reference.

This command applies to the following enterable objects:

- Hierarchical lists
- List boxes
- Subforms (in this case, only an object name — the subform — can be passed in *object*),
- List forms displayed using the **DISPLAY SELECTION** or **MODIFY SELECTION** commands.

If the command is used with an enterable object that is not a list, it then acts the same as the **GOTO OBJECT** command. The command does nothing if the list or the array is empty or invisible. Also, if the list or the array is not enterable, the command only selects the specified item without changing to editing mode. Regarding list boxes, if the column does not allow text entry (entry by check boxes or drop-down lists only), the specified element gets the focus.

The optional *item* parameter allows you to set the position of the item (hierarchical list) or the row number (list box, list forms and subform in “multiple selection” mode) to change to editing mode. If you do not pass this parameter, the command is applied to the current item for *object*. If there is no current item, the first item of *object* changes to editing mode.

**Notes:**

- In subforms and list forms, the command changes the first field of a specified row to edit mode, in the order of entry.
- In listboxes displayed in hierarchical mode, if the targeted item belongs to a hierarchical level that is collapsed, this level (as well as any parent levels) will be expanded automatically so that the item is visible.

## Example 1

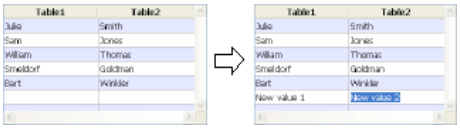
This command can be particularly useful when creating a new item in a hierarchical list. When the command is called, the last item added or inserted in the list automatically becomes editable without the user having to do anything.
The following code may be the method of a button that allows you to insert a new item in an existing list. The default text “New_item” is automatically ready to be changed:

```
vlUniqueRef:=vlUniqueRef+1
INSERT IN LIST(hList;*;"New_item";vlUniqueRef)
EDIT ITEM(*;"MyList")
```


## Example 2

Given two columns in a list box whose variable names are “Array1” and “Array2” respectively. The following example inserts a new item in the two arrays and passes the new item of Array2 into editing mode:

```
$vlRowNum:=Size of array(Array1)+1
LISTBOX INSERT ROWS(*;"MyListBox";$vlRowNum)
Array1{$vlRowNum}:="New value 1"
Array2{$vlRowNum}:="New value 2"
EDIT ITEM(Array2;$vlRowNum)
```



# FILTER KEYSTROKE

FILTER KEYSTROKE ( filteredChar )

| Parameter | Type | | Description |
|---|---|---|---|
| filteredChar | String | ⇒ | Filtered keystroke character or Empty string to cancel the keystroke |

## Description

**FILTER KEYSTROKE** enables you to replace the character entered by the user into a field or an enterable area with the first character of the string *filteredChar* you pass.

If you pass an empty string, the keystroke is cancelled and ignored.

Usually, you will call **FILTER KEYSTROKE** within a form or object method while handling an On Before Keystroke form event. To detect keystroke events, use the command **Form event**. To obtain the actual keystroke, use the commands **Keystroke** or **Get edited text**.

**IMPORTANT NOTE:** The command **FILTER KEYSTROKE** allows you to cancel or replace the character entered by the user with another character. On the other hand, if you want to insert more than one character for a specific keystroke, remember that the text you see on the screen is NOT YET the value of the data source field or variable for the area being edited. The data source field or variable is assigned the entered value after the data entry for the area is validated. It is therefore up to you to "shadow" the data entry into a variable and then to work with this shadow value and reassign the enterable area (see the example in this section). You can also use the **Get edited text** command.

You will use the command **FILTER KEYSTROKE** for:

- Filtering characters in a customized way
- Filtering data entry in a way that you cannot produce using data entry filters
- Implement dynamic lookup or autocomplete areas

**WARNING:** If you call the command **Keystroke** after calling **FILTER KEYSTROKE**, the character you pass to this command is returned instead of the character actually entered.

## Example 1

Using the following code:

```
 //myObject enterable area object method
Case of
   :(Form event=On Load)
      myObject:=""
   :(Form event=On Before Keystroke)
      If(Position(Keystroke;"0123456789")>0)
         FILTER KEYSTROKE("*")
      End if
End case
```

All the digits entered in the area *myObject* are transformed into star characters.

## Example 2

This code implements the behavior of a Password enterable area in which all the entered characters are replaced (on the screen) by random characters:

```
 //vsPassword enterable area object method
Case of
   :(Form event=On Load)
      vsPassword:=""
      vsActualPassword:=""
   :(Form event=On Before Keystroke)
      Handle keystroke(->vsPassword;->vsActualPassword)
      If(Position(Keystroke;Char(Backspace)+Char(Left arrow key)+Char(Right arrow key)+Char(Up arrow key)+Char(Down arrow
```

```
key))=0)
         FILTER KEYSTROKE(Char(65+(Random%26)))
      End if
End case
```

After the data entry is validated, you retrieve the actual password entered by the user in the variable *vsActualPassword*.
Note: The method **Handle keystroke** is listed in the Example section for the command **Keystroke.**

## Example 3

In your application, you have some text areas into which you can enter a few sentences. Your application also includes a dictionary table of terms commonly used throughout your database. While editing your text areas, you would like to be able to quickly retrieve and insert dictionary entries based on the selected characters in a text area. You have two ways to do this:
- Provide some buttons with associated keys, or
- Intercept special keystrokes during the editing of the text area

This example implements the second solution, based on the Help key.

As explained above, during the editing of the text area, the data source for this area will be assigned the entered value after you validate the data entry. In order to retrieve and insert dictionary entries into the text area while this area is being edited, you therefore need to shadow the data entry. You pass pointers to the enterable area and the shadow variable as the first two parameters, and you pass a string of the “forbidden” characters as the third parameter. No matter how the keystroke will be treated, the method returns the original keystroke. The “forbidden” characters are those that you do not want to be inserted into the enterable area and you want to treat as special characters.

```
  ` Shadow keystroke project method
  ` Shadow keystroke ( Pointer ; Pointer ; String ) -> String
  ` Shadow keystroke ( -> srcArea ; -> curValue ; Filter ) -> Old keystroke
C_STRING(1;$0)
C_POINTER($1;$2)
C_TEXT($vtNewValue)
C_STRING(255;$3)
  ` Return the original keystroke
$0:=Keystroke
  ` Get the text selection range within the enterable area
GET HIGHLIGHT($1->;$vlStart;$vlEnd)
  ` Start working with the current value
$vtNewValue:=$2->
  ` Depending on the key pressed or the character entered,
  ` Perform the appropriate actions
Case of
  ` The Backspace (Delete) key has been pressed
   :(Character code($0)=Backspace)
  ` Delete the selected characters or the character at the left of the text cursor
      $vtNewValue:=Delete text($vtNewValue;$vlStart;$vlEnd)
  ` An Arrow key has been pressed
  ` Do nothing, but accept the keystroke
   :(Character code($0)=Left arrow key)
   :(Character code($0)=Right arrow key)
   :(Character code($0)=Up arrow key)
   :(Character code($0)=Down arrow key)

  ` An acceptable character has been entered
   :(Position($0;$3)=0)
      $vtNewValue:=Insert text($vtNewValue;$vlStart;$vlEnd;$0)
   Else
  ` The character is not accepted
      FILTER KEYSTROKE("")
End case
  ` Return the value for the next keystroke handling
$2->:=$vtNewValue
```

This method uses the two following submethods:

```
  ` Delete text project method
  ` Delete text ( String ; Long ; Long ) -> String
  ` Delete text ( -> Text ; SelStart ; SelEnd ) -> New text
```

```
C_TEXT($0;$1)
C_LONGINT($2;$3)
$0:=Substring($1;1;$2-1-Num($2=$3))+Substring($1;$3)

 ` Insert text project method
 ` Insert text ( String ; Long ; Long ; String ) -> String
 ` Insert text ( -> srcText ; SelStart ; SelEnd ; Text to insert ) -> New text
C_TEXT($0;$1;$4)
C_LONGINT($2;$3)
$0:=$1
If($2#$3)
   $0:=Substring($0;1;$2-1)+$4+Substring($0;$3)
Else
   Case of
      :($2<=1)
         $0:=$4+$0
      :($2>Length($0))
         $0:=$0+$4
      Else
         $0:=Substring($0;1;$2-1)+$4+Substring($0;$2)
   End case
End if
```

After you have added these project methods to your project, you can use them in this way:

```
 ` vsDescription enterable area object method
Case of
   :(Form event=On Load)
      vsDescription:=""
      vsShadowDescription:=""
 ` Establish the list of the "forbidden" characters to be treated as special keys
 ` ( here, in this example, only the Help Key is filtered)
      vsSpecialKeys:=Char(HelpKey)
   :(Form event=On Before Keystroke)
      $vsKey:=Shadow keystroke(->vsDescription;->vsShadowDescription;vsSpecialKeys)
      Case of
         :(Character code($vsKey)=Help key)
 ` Do something when the Help key is pressed
 ` Here, in this example, a Dictionary entry must be searched and inserted
            LOOKUP DICTIONARY(->vsDescription;->vsShadowDescription)
      End case
End case
```

The **LOOKUP DICTIONARY** project method is listed below. Its purpose is to use the shadow variable for reassigning the enterable area being edited:

```
 ` LOOKUP DICTIONARY project method
 ` LOOKUP DICTIONARY ( Pointer ; Pointer )
 ` LOOKUP DICTIONARY ( -> Enterable Area ; ->ShadowVariable )

C_POINTER($1;$2)
C_LONGINT($vlStart;$vlEnd)

 ` Get the text selection range within the enterable area
GET HIGHLIGHT($1->;$vlStart;$vlEnd)
 ` Get the selected text or the word on the left of the text cursor
$vtHighlightedText:=Get highlighted text($2->;$vlStart;$vlEnd)
 ` Is there something to look for?
If($vtHighlightedText#"")
 ` If the text selection was the text cursor,
 ` the selection now starts at the word preceeding the text cursor
   If($vlStart=$vlEnd)
      $vlStart:=$vlStart-Length($vtHighlightedText)
   End if
 ` Look for the first avaliable dictionary entry
   QUERY([Dictionary];[Dictionary]Entry=$vtHighlightedText+"@")
 ` Is there one?
   If(Records in selection([Dictionary])>0)
 ` If so, insert it in the shadow text
```

```
      $2->:=Insert text($2->;$vlStart;$vlEnd;[Dictionary]Entry)
   ` Copy the shadow text to the enterable being edited
      $1->:=$2->
   ` Set the selection just after the insert dictionary entry
      $vlEnd:=$vlStart+Length([Dictionary]Entry)
      HIGHLIGHT TEXT(vsComments;$vlEnd;$vlEnd)
   Else
   ` There is no corresponding entry in the Dictionary
      BEEP
   End if
Else
   ` There is no highlighted text
   BEEP
End if
```

The **Get highlighted text** method is listed here:

```
   ` Get highlighted text project method
   ` Get highlighted text ( String ; Long ; Long ) -> String
   ` Get highlighted text ( Text ; SelStart ; SelEnd ) -> highlighted text
C_TEXT($0;$1)
C_LONGINT($2;$3)
If($2<$3)
   $0:=Substring($1;$2;$3-$2)
Else
   $0:=""
   $2:=$2-1
   Repeat
      If($2>0)
         If(Position($1≤$2≥;" ,.!?:;()-_--")=0)
            $0:=$1≤$2≥+$0
            $2:=$2-1
         Else
            $2:=0
         End if
      End if
   Until($2=0)
End if
```

# Get edited text

Get edited text -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Text | Text being entered |

## Description

The **Get edited text** command is mainly to be used with the form event On After Keystroke to retrieve the text as it is being entered. It can also be used with the On Before Keystroke form event. For more information about those form events, please refer to the description of the command **Form event**.

The combination of this command with On Before Keystroke and On After Keystroke form events works as follows:

- As soon as a character is typed on the keyboard, the On Before Keystroke event is generated. In this case, the **Get edited text** function returns the contents of the area before the last keystroke occurred. For example, if the area contains "PA" and the user types an "R", **Get edited text** returns "PA" in the On Before Keystroke event. If the area is empty initially, **Get edited text** returns an empty string.
- Next, the On After Keystroke form event is generated. In this case, **Get edited text** returns the contents of the area including the last character entered on the keyboard. For example, when the area contains "PA" and the user types an "R", **Get edited text** returns "PAR" in the On After Keystroke event.

These two events are only generated in the object methods concerned.

When used in a context other than text entry in a form object, this function returns an empty string.

## Example 1

The following method automatically puts the characters being entered in capitals:

```
If(Form event=On After Keystroke)
   [Trips]Agencies:=Uppercase(Get edited text)
End if
```

## Example 2

Here is an example of how to process on the fly characters entered in a text field. The idea consists of placing in another text field (called "Words") all the words of the sentence being entered. To do so, write the following code in the object method of the field:

```
If(Form event=On After Keystroke)
   $RealTimeEntry:=Get edited text
   PLATFORM PROPERTIES($platform)
   If($platform#3) ` Mac OS
      Repeat
         $DecomposedSentence:=Replace string($RealTimeEntry;Char(32);Char(13))
      Until(Position(" ";$DecomposedSentence)=0)
   Else ` Windows
      Repeat
         $DecomposedSentence:=Replace string($RealTimeEntry;Char(32);Char(13)+Char(10))
      Until(Position(" ";$DecomposedSentence)=0)
   End if
   [Example]Words:=$DecomposedSentence
End if
```

**Note:** This example is not comprehensive because we have assumed that words are separated uniquely by spaces (Char (32)). For a complete solution you will need to add other filters to extract all the words (delimited by commas, semi-colons, apostrophes, etc.).

# GET HIGHLIGHT

GET HIGHLIGHT ( {* ;} object ; startSel ; endSel )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Field, Variable, Form object | ➡ | Object name (if * is specified) or Field or variable (if * is omitted) |
| startSel | Longint | ⬅ | Current text selection starting position |
| endSel | Longint | ⬅ | Current text selection ending position |

## Description

The **GET HIGHLIGHT** command is used to determine what text is currently highlighted in *object*.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass the * parameter, you indicate that the *object* parameter is a field or variable. In this case, you pass the field or variable reference (form fields or variables only) instead of a string.

**Note:** This command cannot be used with fields in the List form of a subform.

Text can be highlighted by the user or by the **HIGHLIGHT TEXT** command.

The parameter *startSel* returns the position of the first highlighted character.

The parameter *endSel* returns the position of the last highlighted character plus one.

If *startSel* and *endSel* are returned equal, the insertion point is positioned before the character specified by *startSel*. The user has not selected any text, and no characters are highlighted.

If the object designated by the *object* parameter is not found in the form, the command returns -1 in *startSel* and -2 in *endSel*.

## Example 1

The following example gets the highlighted selection from the field called *[Products]Comments*:

```
GET HIGHLIGHT([Products]Comments;vFirst;vLast)
If(vFirst<vLast)
   ALERT("The selected text is: "+Substring([Products]Comments;vFirst;vLast-vFirst))
End if
```

## Example 2

See example for the **FILTER KEYSTROKE** command.

## Example 3

Modification of highlighted text style:

```
GET HIGHLIGHT(*;"myText";$startsel,$endsel)
ST SET ATTRIBUTES(*;"myText";$startsel,$endsel;Attribute underline style;1;Attribute bold style;1)
```

# GOTO OBJECT

GOTO OBJECT ( {* ;} object )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified = object is an object name (string) If omitted = object is a field or a variable |
| object | Field, Variable | ⇒ | Object name (if * specified) or Field or Variable (if * omitted) to go to |

## Description

The **GOTO OBJECT** command is used to select the data entry object *object* as the active area of the form. It is equivalent to the user's clicking on or tabbing into the field or variable.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section **Object Properties**.

To remove any focus in the current form, call the command while passing an empty object name in *object* (see example 2).

The **GOTO OBJECT** command can be used in the context of a subform. When it is called from a subform, it first looks for the object in the subform, then, if the search does not find anything there, it extends the search to objects of the parent form.

## Example 1

The **GOTO OBJECT** command can be used in both ways:

```
GOTO OBJECT([People]Name) ` Field Reference
GOTO OBJECT(*;"AgeArea") ` Object Name
```

## Example 2

You don't want any object of the form to have the focus:

```
GOTO OBJECT(*;"")
```

## Example 3

See the example for the **REJECT** command.

# HIGHLIGHT TEXT

HIGHLIGHT TEXT ( {* ;} object ; startSel ; endSel )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Field, Variable, Form object | ⇒ | Object name (if * is specified) or Enterable field or variable (if * is omitted) |
| startSel | Longint | ⇒ | New text selection starting position |
| endSel | Longint | ⇒ | New text selection ending position |

## Description

The **HIGHLIGHT TEXT** command highlights a section of the text in *object*.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass the * parameter, you indicate that the *object* parameter is a field or variable. In this case, you pass the field or variable reference (form fields or variables only) instead of a string.

If *object* is not the object currently being edited, it gets the focus.

**Note:** This command cannot be used with fields in a subform.

*startSel* is the first character position to be highlighted, and *lastSel* is the last character plus one to be highlighted. If *startSel* and *lastSel* are equal, the insertion point is positioned before the character specified by *startSel*, and no characters are highlighted.

If *lastSel* is greater than the number of characters in *object*, then all characters between *startSel* and the end of the text are highlighted.

## Example 1

The following example selects all the characters of the enterable field *[Products]Comments*:

```
HIGHLIGHT TEXT([Products]Comments;1;Length([Products]Comments)+1)
```

## Example 2

The following example moves the insertion point to the beginning of the enterable field *[Products]Comments*:

```
HIGHLIGHT TEXT([Products]Comments;1;1)
```

## Example 3

The following example moves the insertion point to the end of the enterable field *[Products]Comments*:

```
$vLen:=Length([Products]Comments)+1
HIGHLIGHT TEXT([Products]Comments;$vLen;$vLen)
```

## Example 4

See example for the **FILTER KEYSTROKE** command.

# Keystroke

Keystroke -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | String | Character entered by user |

## Description

**Keystroke** returns the character entered by the user into a field or an enterable area.

Usually, you will call **Keystroke** within a form or object method while handling an On Before Keystroke or On After Keystroke form event. To detect keystroke events, use the command **Form event**.

To replace the character actually entered by the user with another character, use the command **FILTER KEYSTROKE**.

**Note:** The **Keystroke** function does not work in subforms.

**IMPORTANT NOTE:** If you want to perform some "on the fly" operations depending on the current value of the enterable area being edited, as well as the new character to be entered, remember that the text you see on screen is NOT YET the value of the data source field or variable for the area being edited. The data source field or variable is assigned the entered value after the data entry for the area is validated (e.g., tabulation to another area, click on a button, and so on). It is therefore up to you to "shadow" the data entry into a variable and then to work with this shadow value. You must do so if you need to know the current text value for executing any particular actions. You can also use the function **Get edited text**.

You will use the command **Keystroke** for:

- Filtering characters in a customized way
- Filtering data entry in a way that you cannot produce using data entry filters
- Implement dynamic lookup or autocomplete areas

## Example 1

See examples for the **FILTER KEYSTROKE** command.

## Example 2

When you process an On Before Keystroke event, you are dealing with the editing of the current text area (the one where the cursor is), not with the "future value" of the data source (field or variable) for this area. The ***Handle keystroke*** project method allows to shadow any text area data entry into a second variable, which you can use to perform the actions while entering characters into the area. You pass a pointer to the area's data source as the first parameter and a pointer to the shadow variable as second parameter. The method returns the new value of the text area in the shadow variable, and returns **True** if the value is different from it what was before the last entered character was inserted.

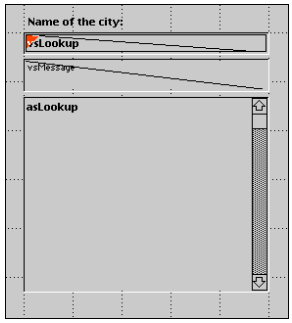
```
  ` Handle keystroke project method
  ` Handle keystroke ( Pointer ; Pointer ) -> Boolean
  ` Handle keystroke ( -> srcArea ; -> curValue ) -> Is new value

C_POINTER($1;$2)
C_TEXT($vtNewValue)

  ` Get the text selection range within the enterable area
GET HIGHLIGHT($1->;$vlStart;$vlEnd)
  ` Start working with the current value
$vtNewValue:=$2->
  ` Depending on the key pressed or the character entered,
  ` Perform the appropriate actions
Case of

  ` The Backspace (Delete) key has been pressed
   :(Character code(Keystroke)=Backspace)
  ` Delete the selected characters or the character at the left of the text cursor
      $vtNewValue:=Substring($vtNewValue;1;$vlStart-1-Num($vlStart=$vlEnd))
      +Substring($vtNewValue;$vlEnd)
```

```
   ` An acceptable character has been entered
    :(Position(Keystroke;"abcdefghjiklmnopqrstuvwxyz -0123456789")>0)
       If($vlStart#$vlEnd)
   ` One or several characters are selected, the keystroke is going to override them
          $vtNewValue:=Substring($vtNewValue;1;$vlStart-1)
          +Keystroke+Substring($vtNewValue;$vlEnd)
       Else
   ` The text selection is the text cursor
          Case of
   ` The text cursor is currently at the begining of the text
             :($vlStart<=1)
   ` Insert the character at the begining of the text
                $vtNewValue:=Keystroke+$vtNewValue
   ` The text cursor is currently at the end of the text
             :($vlStart>=Length($vtNewValue))
   ` Append the character at the end of the text
                $vtNewValue:=$vtNewValue+Keystroke
             Else
   ` The text cursor is somewhere in the text, insert the new character
                $vtNewValue:=Substring($vtNewValue;1;$vlStart-1)+Keystroke
                +Substring($vtNewValue;$vlStart)
          End case
       End if

   ` An Arrow key has been pressed
   ` Do nothing, but accept the keystroke
    :(Character code(Keystroke)=Left arrow key)
    :(Character code(Keystroke)=Right arrow key)
    :(Character code(Keystroke)=Up arrow key)
    :(Character code(Keystroke)=Down arrow key)
   `
    Else
   ` Do not accept characters other than letters, digits, space and dash
       FILTER KEYSTROKE("")
End case
   ` Is the value now different?
$0:=($vtNewValue#$2->)
   ` Return the value for the next keystroke handling
$2->:=$vtNewValue
```

After this project method is added to your application, you can use it as follows:

```
   ` myObject enterable area object method
Case of
   :(Form event=On Load)
      MyObject:=""
      MyShadowObject:=""
   :(Form event=On Before Keystroke)
      If(Handle keystroke(->MyObject;->MyShadowObject))
   ` Perform appropriate actions using the value stored in MyShadowObject
      End if
End case
```

Let's examine the following part of a form:

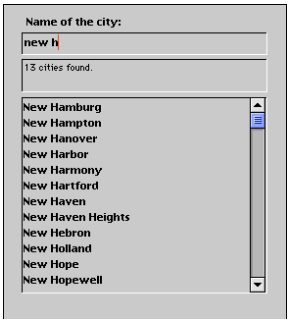



It is composed of the following objects: an enterable area *vsLookup*, a non-enterable area *vsMessage*, and a scrollable area *asLookup*. While entering characters in *vsLookup*, the method for that object performs a query on a [US Zip Codes] table, allowing the user to find US cities by typing only the first characters of the city names.

The *vsLookup* object method is listed here:

```
   ` vsLookup enterable area object method
Case of
   :(Form event=On Load)
      vsLookup:=""
      vsResult:=""
      vsMessage:="Enter the first characters of the city you are looking for."
      CLEAR VARIABLE(asLookup)
   :(Form event=On Before Keystroke)
      If(Handle keystroke(->vsLookup;->vsResult))
         If(vsResult#"")
            QUERY([US Zip Codes];[US Zip Codes]City=vsResult+"@")
            MESSAGES OFF
            DISTINCT VALUES([US Zip Codes]City;asLookup)
            MESSAGES ON
            $vlResult:=Size of array(asLookup)
            Case of
               :($vlResult=0)
                  vsMessage:="No city found."
               :($vlResult=1)
                  vsMessage:="One city found."
               Else
                  vsMessage:=String($vlResult)+" cities found."
            End case
         Else
            DELETE FROM ARRAY(asLookup;1;Size of array(asLookup))
            vsMessage:="Enter the first characters of the city you are looking for."
         End if
      End if
End case
```

Here is the form being executed:




Using the interprocess communication capabilities of 4D, you can similarily build user interfaces in which Lookup features are provided in floating windows that communicate with processes in which records are listed or edited.

# Form Events

- Activated
- After
- Before
- CALL SUBFORM CONTAINER
- Clickcount
- Contextual click
- Deactivated
- Form event Updated 16.0
- In break
- In footer
- In header
- Is waiting mouse up New 16.0
- Outside call
- Right click
- SET TIMER
- *_o_During*

# Activated

Activated -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | Returns TRUE if the execution cycle is an activation |

## Description

The **Activated** command (obsolete) returns **True** in a form method when the window containing the form becomes the frontmost window of the frontmost process.

**Note:** This command is equivalent to using **Form event** and testing whether it returns the On Activate event.

**WARNING:** Do not place a command such as **TRACE** or **ALERT**in the **Activated** phase of the form, as this will cause an endless loop.

**Note:** In order for the **Activated** execution cycle to be generated, make sure that the On Activate event property of the form has been selected in the Design environment.

# After

After -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | Returns True if the execution cycle is an after |

## Description

**After** returns True for the After execution cycle.

In order for the **After** execution cycle to be generated, make sure that the On Validate event property for the form and/or the objects has been selected in the Design environment.

**Note:** This command is equivalent to using **Form event** and testing whether it returns the On Validate event.

# Before

Before -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | Returns True if the execution cycle is a before |

## Description

**Before** returns True for the Before execution cycle.

In order for the **Before** execution cycle to be generated, make sure that the On Load event property for the form and/or the objects has been selected in the Design environment.

**Note:** This command is equivalent to using **Form event** and testing whether it returns the On Load event.

# CALL SUBFORM CONTAINER

CALL SUBFORM CONTAINER ( event )

| Parameter | Type | | Description |
|---|---|---|---|
| event | Longint | ⇨ | Event to be sent |

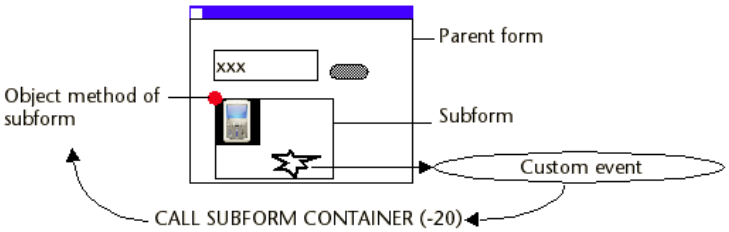
## Description

The **CALL SUBFORM CONTAINER** command lets a subform instance send the *event* to the subform object that contains it. The subform object can then process the *event* in the context of the parent form.

This command must be placed in the form method of the subform or in the object method of one of the subform objects. The event will only be received in the object method of the subform container.

In *event*, you can pass any predefined form event of 4D (you can use the constants of the "**Form Events**" theme) or any value corresponding to a custom event. In the first case, the event must be checked for the subform. In the case of a custom event, it is recommended to pass a negative value in *event* in order to avoid the risk of interfering with existing or future 4D event numbers.

*Example of execution of the* **CALL SUBFORM CONTAINER** *command:*



# Clickcount

Clickcount -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Longint | Number of consecutive clicks |

## Description

The **Clickcount** command returns, in the context of a mouse click event, the number of times the user has clicked in rapid succession with the same mouse button. Typically, this command returns 2 for a double-click.

This command allows you to detect double-clicks in listbox headers or footers, and also to handle sequences of triple-clicks or more.

Every mouse button click generates a separate click event. For example, if a user double-clicks, an event is sent for the first click in which **Clickcount** returns 1; then another event is sent for the second click in which **Clickcount** returns 2.

This command must only be used in the context of On Clicked, On Header Click or On Footer Click form events. It is therefore necessary to check in Design mode that the appropriate event has been properly selected in the Form properties and/or for the specific object.

When both On Clicked and On Double Clicked form events are enabled, the following sequence will be returned by **Clickcount**:

- 1 in On Clicked event
- 2 in On Double Clicked event
- 2+n in On Clicked event

## Example 1

The following code structure can be placed in a listbox header to handle single- and double-clicks:

```
Case of
   :(Form event=On Header Click)
      Case of
         :(Clickcount=1)
            ... //single-click action
         :(Clickcount=2)
            ... //double-click action
      End case
End case
```

## Example 2

Labels are not enterable but they become so after a triple-click. If you want to allow users to edit labels, you can write the following in the object method:

```
If(Form event=On Clicked)
   Case of
      :(Clickcount=3)
         OBJECT SET ENTERABLE(*;"Label";True)
         EDIT ITEM(*;"Label")
   End case
End if
```

# Contextual click

Contextual click -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | True if a contextual click was detected, otherwise False |

## Description

The **Contextual click** command returns True if a contextual click has been made:

- Under Windows and Mac OS, contextual clicks are made using the right button of the mouse.
- Under Mac OS, contextual clicks can also be made using a **Control+click** combination.

This command should be used only in the context of the On Clicked form event. It is therefore necessary to verify in Design mode that the event has been properly selected in the Form properties and/or in the specific object.

## Example

This method, combined with a scrollable area, enables you to change the value of an array element using a context menu:

```
If(Contextual click)
   If(Pop up menu("True:False")=1)
      myArray{myArray}:="True"
   Else
      myArray{myArray}:="False"
   End if
End if
```

# Deactivated

Deactivated -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | Returns TRUE if the execution cycle is a deactivation |

## Description

The **Deactivated** command returns True in a form or object method when the frontmost window of the frontmost process, containing the form, moves to the back.

In order for the **Deactivated** execution cycle to be generated, make sure that the On Deactivate event property of the form and/or the objects has been selected in Design environment.

**Note:** This command is equivalent to using **Form event** and testing whether it returns the On Deactivate event.

# Form event

Form event -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Form event number |

## Description

**Form event** returns a numeric value identifying the type of form event that has just occurred. Usually, you will use **Form event** from within a form or object method.

4D provides predefined constants (found in the **Form Events** theme) in order to compare the values returned by the **Form event** command.

Certain events are generic (generated for any type of object) and others are specific to a particular type of object.

| Constant | Type | Value | Comment |
|---|---|---|---|
| On Load | Longint | 1 | The form is about to be displayed or printed |
| On Mouse Up | Longint | 2 | (*Pictures only*) The user has just released the left mouse button in a Picture object |
| On Validate | Longint | 3 | The record data entry has been validated |
| On Clicked | Longint | 4 | A click occurred on an object |
| On Header | Longint | 5 | The form's header area is about to be printed or displayed |
| On Printing Break | Longint | 6 | One of the form's break areas is about to be printed |
| On Printing Footer | Longint | 7 | The form's footer area is about to be printed |
| On Display Detail | Longint | 8 | A record is about to be displayed in a list or a row is about to be displayed in a list box. |
| On Outside Call | Longint | 10 | The form received a **CALL PROCESS** call |
| On Activate | Longint | 11 | The form's window becomes the frontmost window |
| On Deactivate | Longint | 12 | The form's window ceases to be the frontmost window |
| On Double Clicked | Longint | 13 | A double click occurred on an object |
| On Losing Focus | Longint | 14 | A form object is losing the focus |
| On Getting Focus | Longint | 15 | A form object is getting the focus |
| On Drop | Longint | 16 | Data has been dropped onto an object |
| On Before Keystroke | Longint | 17 | A character is about to be entered in the object that has the focus. **Get edited text** returns the object's text without this character. |
| On Menu Selected | Longint | 18 | A menu item has been chosen |
| On Plug in Area | Longint | 19 | An external object requested its object method to be executed |
| On Data Change | Longint | 20 | Object data has been modified |
| On Drag Over | Longint | 21 | Data could be dropped onto an object |
| On Close Box | Longint | 22 | The window's close box has been clicked |
| On Printing Detail | Longint | 23 | The form's detail area is about to be printed |
| On Unload | Longint | 24 | The form is about to be exited and released |
| On Open Detail | Longint | 25 | The detail form associated with the output form or with the listbox is about to be opened |
| On Close Detail | Longint | 26 | You left the detail form and are going back to the output form |
| On Timer | Longint | 27 | The number of ticks defined by the **SET TIMER** command has passed |
| On After Keystroke | Longint | 28 | A character is about to be entered in the object that has the focus. **Get edited text** returns the object's text including this character. |
| On Resize | Longint | 29 | The form window is resized |
| On After Sort | Longint | 30 | (*List box only*) A standard sort has just been carried out in a list box column |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| On Selection Change | Longint | 31 | • *List box*: The current selection of rows or columns is modified<br>• *Records in list*: The current record or the current selection of rows is modified in a list form or subform<br>• *Hierarchical list*: The selection in the list is modified following a click or a keystroke<br>• *Enterable field or variable*: The text selection or the position of the cursor in the area is modified following a click or a keystroke |
| On Column Moved | Longint | 32 | (*List box only*) A list box column is moved by the user via drag and drop |
| On Column Resize | Longint | 33 | (*List box only*) The width of a list box column is modified by a user with the mouse |
| On Row Moved | Longint | 34 | (*List box only*) A list box row is moved by the user via drag and drop |
| On Mouse Enter | Longint | 35 | The mouse cursor enters the graphic area of an object |
| On Mouse Leave | Longint | 36 | The mouse cursor leaves the graphic area of an object |
| On Mouse Move | Longint | 37 | The mouse cursor moves at least one pixel OR a modifier key (Shift, Alt, Shift Lock) was pressed. If the event is checked for an object only, it is generated only when the cursor is within the graphic area of the object |
| On Alternative Click | Longint | 38 | • *3D buttons*: The “arrow” area of a 3D button is clicked<br>• *List boxes*: In a column of an object array, an ellipsis button ("alternateButton" attribute) is clicked<br>**Note**: Ellipsis buttons are only available for versions v15 or higher. |
| On Long Click | Longint | 39 | (*3D buttons only*) A 3D button is clicked and the mouse button remains pushed for a certain lapse of time |
| On Load Record | Longint | 40 | During entry in list, a record is loaded during modification (the user clicks on a record line and a field changes to editing mode) |
| On Before Data Entry | Longint | 41 | (*List box only*) A list box cell is about to change to editing mode |
| On Header Click | Longint | 42 | (*List box only*) A click occurs in a column header of the list box |
| On Expand | Longint | 43 | (*Hierarchical lists and hierarchical list boxes*) An element of the hierarchical list or hierarchical list box has been expanded using a click or a keystroke |
| On Collapse | Longint | 44 | (*Hierarchical lists and hierarchical list boxes*) An element of the hierarchical list or hierarchical list box has been collapsed using a click or a keystroke |
| On After Edit | Longint | 45 | The contents of the enterable object that has the focus has just been modified |
| On Begin Drag Over | Longint | 46 | An object is being dragged |
| On Begin URL Loading | Longint | 47 | (*Web areas only*) A new URL is loaded in the Web area |
| On URL Resource Loading | Longint | 48 | (*Web areas only*) A new resource is loaded in the Web area |
| On End URL Loading | Longint | 49 | (*Web areas only*) All the resources of the URL have been loaded |
| On URL Loading Error | Longint | 50 | (*Web areas only*) An error occurred when the URL was loading |
| On URL Filtering | Longint | 51 | (*Web areas only*) A URL was blocked by the Web area |

| Constant | Type | Value | Comment |
|---|---|---|---|
| On Open External Link | Longint | 52 | (*Web areas only*) An external URL has been opened in the browser |
| On Window Opening Denied | Longint | 53 | (*Web areas only*) A pop-up window has been blocked |
| On bound variable change | Longint | 54 | The variable bound to a subform is modified. |
| _o_On Mac toolbar button | Longint | 55 | *** Obsolete constant *** |
| On Page Change | Longint | 56 | The current page of the form is changed |
| On Footer Click | Longint | 57 | (*List boxes only*) A click occurs in the footer of a list box or a list box column |
| On Delete Action | Longint | 58 | (*Hierarchical lists and List boxes*) The user attempts to delete an item |
| On Scroll | Longint | 59 | The user scrolls the contents of a picture field or variable using the mouse or keyboard. |

**Note:** The events specific to output forms cannot be used with **project forms**. This includes: On Display Detail, On Open Detail, On Close Detail, On Load Record, On Header, On Printing Detail, On Printing Break, On Printing Footer.

## Events and Methods

When a form event occurs, 4D performs the following actions:

- First, it browses the objects of the form and calls the object method for any object (involved in the event) whose corresponding object event property has been selected.
- Second, it calls the form method if the corresponding form event property has been selected.

Do not assume that the object methods, if any, will be called in a particular order. The rule of thumb is that the object methods are always called before the form method. If an object is a subform, the object methods of the subform's list form are called, then the form method of the list form is called. 4D then continues to call the object methods of the parent form. In other words, when an object is a subform, 4D uses the same rule of thumb for the object and form methods within the subform object.

Except for the On Load and On Unload events, if the form event property is not selected for a given event, this does not prevent calls to object methods for the objects whose same event property is selected. In other words, enabling or disabling an event at the form level has no effect on the object event properties.

The number of objects involved in an event depends on the nature of the event:

- On Load event - All the objects of the form (from any page) whose On Load object event property is selected will have their object method called. Then, if the On Load form event property is selected, the form will have its form method called.
- On Activate or On Resize event - No object method will be called, because this event applies to the form as a whole and not to a particular object. Consequently, if the On Activate form event property is selected, only the form will have its form method called.
- On Timer event - This event is generated only if the form method contains a previous call to the **SET TIMER** command. If the On Timer form event property is selected, only the form method will receive the event, no object method will be called.
- On Drag Over event - Only the droppable object involved in the event will have its object method called if the "Droppable" event property is selected for it. The form method will not be called.
- Conversely, for the On Begin Drag Over event, the object method or form method of the object being dragged will be called (if the "Draggable" event property is selected for the it).

**WARNING:** Unlike all other events, during a On Begin Drag Over event, the method called is executed in the context of the process of the drag and drop source object, not in that of the drag and drop destination object. For more information, see the **Drag and Drop** section.

- If the On Mouse Enter, On Mouse Move and On Mouse Leave events have been checked for the form, they are generated for each form object. If they are checked for an object, they are generated only for that object. When there are superimposed objects, the event is generated by the first object capable of managing it that is found going in order from top level to bottom. Objects that are made invisible using the **OBJECT SET VISIBLE** command do not generate these events. During object entry, other objects may receive these type of events depending on the position of the mouse.
  Note that the On Mouse Move event is generated not only when the mouse cursor is moved but also when the user presses a modifier key such as **Shift**, **Shift Lock**, **Ctrl** or **Option** (this makes it possible to manage copy- or move-type drag-and-drop operations).

- Records in list: The sequence of calls to methods and form events in the list forms displayed via **DISPLAY SELECTION** and **MODIFY SELECTION** and the subforms is as follows:
  For each object in the header area:
    - Object method with On Header event
    - Form method with On Header event

  For each record:
    - For each object in the detail area:
      - Object method with On Display Detail event
    - Form method with On Display Detail event

- Calling a 4D command that displays a dialog box from the On Display Detail and On Header events is not allowed and will cause a syntax error to occur. More particularly, the commands concerned are: **ALERT**, **DIALOG**, **CONFIRM**, **Request**, **ADD RECORD**, **MODIFY RECORD**, **DISPLAY SELECTION** and **MODIFY SELECTION**.
- On Page Change: This event is only available at the form level (it is called in the form method). It is generated each time the current page of the form changes (following a call to the **FORM GOTO PAGE** command or a standard navigation action). Note that it is generated after the page is fully loaded, i.e. once all the objects it contains are initialized (including Web areas). This event is useful for executing code that requires all objects to be initialized beforehand. You can also use it to optimize the application by executing code (for example, a search) only after a specific page of the form is displayed and not just as soon as page 1 is loaded. If the user does not go to this page, the code is not executed .

The following table summarizes how object and form methods are called for each event type:

| Event | Object Methods | Form Method | Which Objects |
|---|---|---|---|
| On Load | Yes | Yes | All objects |
| On Unload | Yes | Yes | All objects |
| On Validate | Yes | Yes | All objects |
| On Clicked | Yes (if clickable or enterable) (*) | Yes | Involved object only |
| On Double Clicked | Yes (if clickable or enterable) (*) | Yes | Involved object only |
| On Before Keystroke | Yes (if enterable) (*) | Yes | Involved object only |
| On After Keystroke | Yes (if enterable) (*) | Yes | Involved object only |
| On After Edit | Yes (if enterable) (*) | Yes | Involved object only |
| On Getting Focus | Yes (if tabbable) (*) | Yes | Involved object only |
| On Losing Focus | Yes (if tabbable) (*) | Yes | Involved object only |
| On Activate | Never | Yes | None |
| On Deactivate | Never | Yes | None |
| On Outside Call | Never | Yes | None |
| On Page Change | Never | Yes | None |
| On Begin Drag Over | Yes (if draggable) (**) | Yes | Involved object only |
| On Drop | Yes (if droppable) (**) | Yes | Involved object only |
| On Drag Over | Yes (if droppable) (**) | Never | Involved object only |
| On Mouse Enter | Yes | Yes | All objects |
| On Mouse Move | Yes | Yes | All objects |
| On Mouse Leave | Yes | Yes | All objects |
| On Mouse Up | Yes | Never | Involved object only |
| On Menu Selected | Never | Yes | None |
| On Bound variable change | Never | Yes | None |
| On Data Change | Yes (if modifiable) (*) | Yes | Involved object only |
| On Plug in Area | Yes | Yes | Involved object only |
| On Header | Yes | Yes | All objects |
| On Printing Detail | Yes | Yes | All objects |
| On Printing Break | Yes | Yes | All objects |
| On Printing Footer | Yes | Yes | All objects |
| On Close Box | Never | Yes | None |
| On Display Detail | Yes | Yes | All objects |
| On Open Detail | No, except for List boxes | Yes | None except List boxes |
| On Close Detail | No, except for List boxes | Yes | None except List boxes |
| On Resize | Never | Yes | None |
| On Selection Change | Yes (***) | Yes | Involved object only |
| On Load Record | Never | Yes | None |
| On Timer | Never | Yes | None |
| On Scroll | Yes | Never | Involved object only |
| On Before Data Entry | Yes (List box) | Never | Involved object only |
| On Column Moved | Yes (List box) | Never | Involved object only |
| On Row Moved | Yes (List box) | Never | Involved object only |
| On Column Resize | Yes (List box) | Never | Involved object only |
| On Header Click | Yes (List box) | Never | Involved object only |
| On Footer Click | Yes (List box) | Never | Involved object only |
| On After Sort | Yes (List box) | Never | Involved object only |
| On Long Click | Yes (3D button) | Yes | Involved object only |
| On Alternative Click | Yes (3D button and List box) | Never | Involved object only |
| On Expand | Yes (Hier. list and list box) | Never | Involved object only |
| On Collapse | Yes (Hier. list and list box) | Never | Involved object only |
| On Delete Action | Yes (Hier. list and list box) | Never | Involved object only |
| On URL Resource Loading | Yes (Web Area) | Never | Involved object only |

| | | | |
|---|---|---|---|
| On Begin URL Loading | Yes (Web Area) | Never | Involved object only |
| On URL Loading Error | Yes (Web Area) | Never | Involved object only |
| On URL Filtering | Yes (Web Area) | Never | Involved object only |
| On End URL Loading | Yes (Web Area) | Never | Involved object only |
| On Open External Link | Yes (Web Area) | Never | Involved object only |
| On Window Opening Denied | Yes (Web Area) | Never | Involved object only |

(*) For more infomation, see the "Events, Objects and Properties" section below.
(**) Refer to the "**Drag and Drop**" chapter for more information.
(***) Only list box, hierarchical list and subform type objects support this event.

**IMPORTANT:** Always keep in mind that, for any event, the method of a form or an object is called if the corresponding event property is selected for the form or objects. The benefit of disabling events in the Design environment (using the Property List of the Form editor) is that you can greatly reduce the number of calls to methods and therefore significantly optimize the execution speed of your forms.

**WARNING:** The On Load and On Unload events are generated for objects if they are enabled for both the objects and the form to which the objects belong. If the events are enabled for objects only, they will not occur; these two events must also be enabled at the form level.

## Events, Objects and Properties

An object method is called if the event can actually occur for the object, depending on its nature and properties. The following section details the events you will generally use to handle the various types of objects.
Keep in mind that the Property List of the Form editor only displays the events compatible with the selected object or the form.

### Clickable Objects

Clickable objects are mainly handled using the mouse. They include:

- Boolean enterable fields or variables
- Buttons, default buttons, radio buttons, check boxes, button grids
- 3D Buttons, 3D radio buttons, 3D check boxes
- Pop-up menus, hierarchical pop-up menus, picture menus
- Drop-down lists, menus/drop-down lists
- Scrollable areas, hierarchical lists, list boxes and list box columns
- Invisible buttons, highlight buttons, radio pictures
- Thermometers, rulers, dials (also known as slider objects)
- Tab controls
- Splitters.

After the On Clicked or On Double Clicked object event property is selected for one of these objects, you can detect and handle the clicks within or on the object, using the **Form event** command that returns On Clicked or On Double Clicked, depending on the case.

**Note:** Starting with 4D v14, enterable fields and variables containing text (of the text, date, time, or number type) also generate the On Clicked and On Double Clicked events.

If both events are selected for an object, the On Clicked and then the On Double Clicked events will be generated when the user double-clicks the object.

For all these objects, the On Clicked event occurs once the mouse button is released. However, there are several exceptions:

- Invisible buttons - The On Clicked event occurs as soon as the click is made and does not wait for the mouse button to be released.
- Slider objects (thermometers, rulers, and dials) - If the display format indicates that the object method must be called while you are sliding the control, the On Clicked event occurs as soon as the click is made.

In the context of an On Clicked event, you can test the number of clicks made by the user by means of the **Clickcount** command.

**Note:** Some of these objects can be activated with the keyboard. For example, once a check box gets the focus, it can be entered using the space bar. In such a case, an On Clicked event is still generated.

**WARNING:** Combo boxes are not considered to be clickable objects. A combo box must be treated as an enterable text area whose associated drop-down list provides default values. Consequently, you handle data entry within a combo box through

the On Before Keystroke, On After Keystroke and On Data Change events.

**Note**: Starting with 4D v13, pop-up menu/drop-down list and hierarchical pop-up menus can generate the On Data Change event. This allows you to detect the activation of the object when a value different from the current value is selected.

### Keyboard Enterable Objects

Keyboard enterable objects are objects into which you enter data using the keyboard and for which you may filter the data entry at the lowest level by detecting On After Edit, On Before Keystroke, On After Keystroke and On Selection Change events. You can take advantage of these events using the **Get edited text** command.
Keyboard enterable objects and data types include:

- All enterable field objects of the alpha, text, date, time, number or (On After Edit only) picture type
- All enterable variables of the alpha, text, date, time, number or (On After Edit only) picture type
- Combo boxes (except On Selection Change)
- List boxes.

**Note:** Starting with 4D v14, enterable fields and variables containing text (of the text, date, time, or number type) also generate the On Clicked and On Double Clicked events.

**Note:** Even though they are "enterable" objects, hierarchical lists do not manage the On After Edit, On Before Keystroke and On After Keystroke form events (see also the "Hierarchical lists" paragraph below).

- On Before Keystroke and On After Keystroke
  **Note:** The On After Keystroke event can generally be replaced by the On After Edit event (see below).
  After the On Before Keystroke and On After Keystroke event properties are selected for an object, you can detect and handle the keystrokes within the object, using the Form event command that will return On Before Keystroke and then On After Keystroke (for more information, please refer to the description of the **Get edited text** command). These events are also activated by language commands that simulate a user action like **POST KEY**.
  Keep in mind that user modifications that are not carried out using the keyboard (paste, drag-drop, etc.) are not taken into account. To process these events, you must use On After Edit.
  **Note:** The On Before Keystroke and On After Keystroke events are not generated when using an input method. An input method (or IME, Input Method Editor) is a program or a system component that can be used to enter complex characters or symbols (for example, Japanese or Chinese) using a Western keyboard.

- On After Edit
  When it is used, this event is generated after each change made to the contents of an enterable object, regardless of the action that caused the change, i.e.:
  - Standard editing actions which modify content like paste, cut, delete or cancel;
  - Dropping a value (action similar to paste);
  - Any keyboard entry made by the user; in this case, the On After Edit event is generated after the On Before Keystroke and On After Keystroke events, if they are used.
  - Any modification made using a language command that simulates a user action (i.e., **POST KEY**).

  Be aware that the following actions do NOT trigger this event:
  - Editing actions that do not modify the contents of the area like copy or select all; or dragging a value (action similar to copy); however, these actions do modify the location of the cursor and thus trigger the On Selection Change event.
  - Any modifications made to the contents by programming, except for the commands simulating a user action.
  This event can be used to control user actions in order, for example, to prevent the pasting in of text that is too long, to block certain characters or to prevent a password field from being cut.

- On Selection Change
  When it is applied to a text field or variable (enterable or not), this event is triggered each time the position of the cursor changes. This happens, for instance, as soon as the user selects text using the mouse or keyboard arrow keys, or when the user enters text. This lets you call, for example, commands such as **GET HIGHLIGHT**.

### Modifiable Objects

Modifiable objects have a data source whose value can be changed using the mouse or the keyboard; they are not truly considered as user interface controls handled through the On Clicked event. They include:

- All enterable field objects (except BLOBs)
- All enterable variables (except BLOBs, pointers, and arrays)
- Combo boxes
- External objects (for which full data entry is accepted by the plug-in)
- Hierarchical lists
- List boxes and list box columns.

These objects receive On Data Change events. After the On Data Change object event property is selected for one of these objects, you can detect and handle the change of the data source value, using the **Form event** command that will return On Data Change. The event is generated as soon as the variable associated with the object is updated internally by 4D (i.e., in general, when the entry area of the object loses the focus).

### Tabbable Objects

Tabbable objects get the focus when you use the Tab key to reach them and/or click on them. The object having the focus receives the characters (typed on the keyboard) that are not modifiers to a menu item or to an object such as a button.

All objects are tabbable, EXCEPT the following:

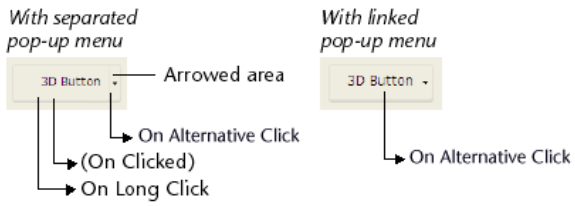
- Non-enterable fields or variables
- Button grids
- 3D buttons, 3D radio buttons, 3D check boxes
- Pop-up menus, hierarchical pop-up menus
- Menus/drop-down lists
- Picture menus
- Scrollable areas
- Invisible buttons, highlight buttons, radio picture buttons
- Graphs
- External objects (for which full data entry is accepted by the 4D plug-in)
- Tab controls
- Splitters.

After the On Getting Focus and/or On losing Focus object event properties are selected for a tabbable object, you can detect and handle the change of focus, using the **Form event** command that will return On Getting Focus or On losing Focus, depending on the case.

### 3D buttons

3D buttons let you set up advanced graphic interfaces (for a description of 3D buttons, refer to the Design Reference manual). In addition to generic events, two specific events can be used to manage these buttons:

- On Long Click: This event is generated when a 3D button receives a click and the mouse button is held for a certain length of time. In theory, the length of time for which this event is generated is equal to the maximum length of time separating a double-click, as defined in the system preferences.
  This event can be generated for all styles of 3D buttons, 3D radio buttons and 3D check boxes, with the exception of “previous generation” 3D buttons (i.e. background offset style) and arrow areas of 3D buttons with a pop-up menu (see below).
  This event is generally used to display pop-up menus in case of long button clicks. The On Clicked event, if enabled, is generated if the user releases the mouse button before the “long click” time limit.
- On Alternative Click: Some 3D button styles can be linked to a pop-up menu and display an arrow. Clicking on this arrow causes a selection pop-up to appear that provides a set of alternative actions in relation to the primary button action.
  4D allows you to manage this type of button using the On Alternative Click event. This event is generated when the user clicks on the “arrow” (as soon as the mouse button is held down):
  - If the pop-up menu is “separated,” the event is only generated when a click occurs on the portion of the button with the arrow.
  - If the pop-up menu is “linked,” the event is generated when a click occurs on any part of the button. Please note that the On Long Click event cannot be generated with this type of button.




The following 3D button, 3D radio button and 3D check box styles accept the “With pop-up menu” property: None, Toolbar button, Bevel, Rounded bevel and Office XP.

### List boxes

Several form events can be used to manage various specific features of list boxes:

- On Before Data Entry: This event is generated just before a cell in the list box is edited (before the entry cursor is displayed). This event allows the developer, for example, to display a different text depending on whether the user is in the display or edit mode.
- On Selection Change: This event is generated each time the current selection of rows or columns of the list box is modified. This event is also generated for lists of records and hierarchical lists.
- On Column Moved: This event is generated when a column of the list box is moved by the user using drag and drop. It is not generated if the column is dragged and then dropped in its initial location. The **LISTBOX MOVED COLUMN NUMBER** command returns the new position of the column.
- On Row Moved: This event is generated when a row of the list box is moved by the user using drag and drop. It is not generated if the row is dragged and then dropped in its initial location.
- On Column Resize: This event is generated when the width of a column in the list box is modified by a user. Starting with 4D v16, the event is triggered "live," i.e., sent continuously during the event, for as long as the list box or column concerned is being resized. This resizing is performed manually by a user, or may occur as a result of the list box and its column(s) being resized along with the form window itself (whether the form is resized manually or using the **RESIZE FORM WINDOW** command).
  **Note:** The On Column Resize event is not triggered when a "fake" column is resized (for more information on fake columns, please refer to **Resizing Options theme**).
- On Expand and On Collapse: These events are generated when a row of the hierarchical list box is expanded or collapsed.
- On Header Click: This event is generated when a click occurs on the header of a column in the list box. In this case, the **Self** command lets you find out the header of the column that was clicked. The On Clicked event is generated when a right click (Windows) or Ctrl+click (Mac OS) occurs on a column or column header. You can test the number of clicks made by the user by means of the **Clickcount** command.
  If the **Sortable** property was checked in the list box, you can decide whether or not to authorize a standard sort of the column by passing the value 0 or -1 in the $0 variable:
  - If $0 equals 0, a standard sort is performed.
  - If $0 equals -1, a standard sort is not performed and the header does not display the sort arrow. The developer can still generate a column sort based on customized sort criteria using the 4D array management commands.
  If the **Sortable** property is not selected for the list box, the $0 variable is not used.
- On Footer Click: This event is available for a list box or list box column object. It is generated when a click occurs in the footer of a list box or a list box column. In this case, the **OBJECT Get pointer** command returns a pointer to the variable of the footer that is clicked. The event is generated for both left and right clicks.
  You can test the number of clicks made by the user by means of the **Clickcount** command.
- On After Sort: This event is generated just after a standard sort is performed (i.e. it is not generated if $0 returns -1 in the On Header Click event). This mechanism is useful for storing the directions of the last sort performed by the user. In this event, the **Self** command returns a pointer to the variable of the sorted column header.
- On Delete Action: This event is generated each time a user attempts to delete the selected item(s) by pressing a deletion key (**Delete** or **Backspace**) or selecting a menu item whose associated standard action is 'Clear' (such as the **Clear** command in the **Edit** menu). This event is only available at the level of the list box object. Note that generating the event is the only action carried out by 4D: the program does not delete any items. It is up to the developer to handle the deletion and any prior warning messages that are displayed.
- On Scroll (new in v15): This event is generated as soon as a user scrolls the rows or columns of the list box. The event is only generated when the scroll is the result of a user action: using the scroll bars and/or cursors, using the mouse wheel or the keyboard. It is not generated when scrolling is due to the execution of the **OBJECT SET SCROLL POSITION** command.
  This event is triggered after any other user event related to the scrolling action (On Clicked, On After Keystroke, etc.). The event is only generated in the object method (not in the form method). Refer to example 15.
- On Alternative Click (new in v15): This event is generated in columns of object array type list boxes, when the user clicks on a widget ellipsis button ("alternateButton" attribute). For more information, refer to the **Using object arrays in columns (4D View Pro)** section.

Two generic events can also be used in the context of "selection" type list boxes:

- On Open Detail: This event is generated when a record is about to be displayed in the detail form associated with a "selection" type list box (and before this form is opened).
- On Close Detail: This event is generated when a record displayed in the detail form associated with a "selection" type list box is about to be closed (regardless of whether or not the record was modified).

### Hierarchical lists

In addition to generic events, several specific events can be used to handle user actions performed on hierarchical lists:

- On Selection Change: This event is generated every time the selection in the hierarchical list is modified after a mouse click or keystroke.
  This event is also generated in list box objects and record lists.
- On Expand: This event is generated every time an element of the hierarchical list is expanded with a mouse click or keystroke.
- On Collapse: This event is generated every time an element of the hierarchical list is collapsed with a mouse click or keystroke.
- On Delete Action: This event is generated each time a user attempts to delete the selected item(s) by pressing a deletion key (**Delete** or **Backspace**) or selecting the **Delete** command from the **Edit** menu. Note that generating the event is the only action carried out by 4D: the program does not delete any items. It is up to the developer to handle the deletion and any prior warning messages that are displayed (see the example).

These events are not mutually exclusive. They can be generated one after another for a hierarchical list:
- Following a keystroke (in order):

| Event | Context |
|---|---|
| On Data Change | Element was edited |
| On Expand/On Collapse | Opening/Closing of a sublist using the -> or <- arrow keys |
| On Selection Change | Selection of a new element |
| On Clicked | Activation of the list using keyboard |

- Following a mouse click (in order):

| Event | Context |
|---|---|
| On Data Change | Element was edited |
| On Expand/On Collapse | Opening/Closing of a sublist using the expand/collapse icons or Double-click on non-editable sublist |
| On Selection Change | Selection of a new element |
| On Clicked / On Double Clicked | Activation of the list using click or double-click |

### Picture fields and variables

- The On Scroll form event is generated as soon as a user scrolls a picture within the area (field or variable) that contains it. You can scroll the contents of a picture area when the size of the area is smaller than its contents and the display format is "**Truncated (non Centered)**". For more information about this, refer to **Picture formats**.
  The event is generated when the scroll is the result of a user action: using the scroll bars and/or cursors, using the mouse wheel or the keyboard (for more information about scrolling using the keyboard, refer to **Scroll bars**). It is not generated when the object is scrolled due to the execution of the **OBJECT SET SCROLL POSITION** command. This event is triggered after any other user event related to the scrolling action (On Clicked, On After Keystroke, etc.). It is generated in the object method (not in the form method). Refer to example 14.

- (New in v16) The On Mouse Up event is generated when the user has just released the left mouse button while dragging in a picture area (field or variable). This event is useful, for example, when you want the user to be able to move, resize or draw objects in a SVG area.
  When the On Mouse Up event is generated, you can get the local coordinates where the mouse button was released. These coordinates are returned in the **MouseX** and **MouseY** **System Variables**. The coordinates are expressed in pixels with respect to the top left corner of the picture (0,0).
  When using this event, you must also use the **Is waiting mouse up** command to handle cases where the "state manager" of the form is desynchronized, i.e. when the On Mouse Up event is not received after a click. This is the case for example when an alert dialog box is displayed above the form while the mouse button has not been released. For more information and an example of use of the On Mouse Up event, please refer to the description of the **Is waiting mouse up** command.

  **Note:** If the "Draggable" option is checked for the picture object, the On Mouse Up event is never generated.

### Subforms

A subform container object (object included in the parent form, which contains one subform instance) supports the following events:

- On Load and On Unload: respectively opening and closing of the subform (these events must also have been activated at the parent form level in order to be taken into account). Note that these events are generated before those of the parent form. Also note that, in accordance with the operating principles of form events, if the subform is placed on a

page other than page 0 or 1, these events will only be generated when that page is displayed/closed (and not when the form is displayed/closed.
- On Validate: validation of data entry in the subform.
- On Data Change: the value of the variable of the subform object has been modified.
- On Getting Focus and On Losing Focus: subform container just got or lost the focus. These events are generated in the method of the subform object when they are checked. They are sent to the form method of the subform, which means, for example, that you can manage the display of navigation buttons in the subform according to the focus.
  Note that subform objects can themselves have the focus.
- On Bound Variable Change: This specific event is generated in the context of the form method of the subform as soon as a value is assigned to the variable bound with the subform in the parent form (even if the same value is reassigned) and if the subform belongs to the current form page or to page 0. For more information about managing subforms, please refer to the *Design Reference* manual.

**Note:** It is possible to specify any custom event type that could be generated in a subform via the **CALL SUBFORM CONTAINER** command. This command lets you call the container object method and to pass an event code to it.

**Note:** The On Clicked and On Double Clicked events generated in the subform are received first by the form method of the subform and then by the form method of the host form.

### Web Areas

Seven form events are available specifically for Web areas:

- On Begin URL Loading: This event is generated at the start of loading a new URL in the Web area. The "URL" variable associated with the Web area can be used to find out the URL being loaded.
  **Note:** The URL being loaded is different from the current URL (refer to the description of the **WA Get current URL** command).
- On URL Resource Loading: This event is generated each time a new resource (picture, frame, etc.) is loaded on the current Web page.
  The "Progression" variable associated with the area lets you find out the current state of the loading.
- On End URL Loading: This event is generated when all the resources of the current URL have been loaded.
  You can call the **WA Get current URL** command in order to find out the URL that was loaded.
- On URL Loading Error: This event is generated when an error is detected during the loading of a URL.
  You can call the **WA GET LAST URL ERROR** command in order to get information about the error.
- On URL Filtering: This event is generated when the loading of a URL is blocked by the Web area because of a filter set up using the **WA SET URL FILTERS** command.
  You can find out the blocked URL using the **WA Get last filtered URL** command.
- On Open External Link: This event is generated when the loading of a URL was blocked by the Web area and the URL was opened with the current system browser, because of a filter set up via the **WA SET EXTERNAL LINKS FILTERS** command.
  You can find out the blocked URL using the **WA Get last filtered URL** command.
- On Window Opening Denied: This event is generated when the opening of a pop-up window is blocked by the Web area. 4D Web areas do not allow the opening of pop-up windows.
  You can find out the blocked URL using the **WA Get last filtered URL** command.

## Example 1

This example shows the On Validate event being used to automatically assign (to a field) the date that the record is modified:

```
 //Method of a form
Case of
 // ...
   :(Form event=On Validate)
      [aTable]Last Modified On:=Current date
End case
```

## Example 2

In this example, the complete handling of a drop-down list (initialization, user clicks, and object release) is encapsulated in the method of the object:

```
 //asBurgerSize Drop-down list Object Method
Case of
   :(Form event=On Load)
      ARRAY TEXT(asBurgerSize;3)
      asBurgerSize{1}:="Small"
      asBurgerSize{1}:="Medium"
      asBurgerSize{1}:="Large"
   :(Form event=On Clicked)
      If(asBurgerSize#0)
         ALERT("You chose a "+asBurgerSize{asBurgerSize}+" burger.")
      End if
   :(Form event=On Unload)
      CLEAR VARIABLE(asBurgerSize)
End case
```

## Example 3

This example shows how, in an object method, to accept and later handle a drag and drop operation for a field object that only accepts picture values.

```
 //[aTable]aPicture enterable picture field object method
Case of
   :(Form event=On Drag Over)
 //A drag-and-drop operation has started and the mouse is currently over the field
 //Get the information about the source object
      DRAG AND DROP PROPERTIES($vpSrcObject;$vlSrcElement;$lSrcProcess)
 //Note that we do not need to test the source process ID number
 //for the object method executed since it is in the same process
      $vlDataType:=Type($vpSrcObject->)
 //Is the source data a picture (field, variable or array)?
      If(($vlDataType=Is picture)|($vlDataType=Picture array))
 //If so, accept the drag.
         $0:=0
      Else
 //If so, refuse the drag
         $0:=-1
      End if
   :(Form event=On Drop)
 //The source data has been dropped on the object, we therefore need to copy it
 //into the object
 //Get the information about the source object
      DRAG AND DROP PROPERTIES($vpSrcObject;$vlSrcElement;$lSrcProcess)
      $vlDataType:=Type($vpSrcObject->)
      Case of
 //The source object is Picture field or variable
         :($vlDataType=Is picture)
 //Is the source object from the same process (thus from the same window and form)?
            If($lSrcProcess=Current process)
 //If so, just copy the source value
               [aTable]aPicture:=$vpSrcObject->
            Else
 //If not, is the source object a variable?
               If(Is a variable($vpSrcObject))
 //If so, get the value from the source process
                  GET PROCESS VARIABLE($lSrcProcess;$vpSrcObject->;$vgDraggedPict)
                  [aTable]aPicture:=$vgDraggedPict
               Else
 //If not, use CALL PROCESS to get the field value from the source process
               End if
            End if
 //The source object is an array of pictures
         :($vlDataType=Picture array)
 //Is the source object from the same process (thus from the same window and form)?
            If($lSrcProcess=Current process)
 //If so, just copy the source value
               [aTable]aPicture:=$vpSrcObject->{$vlSrcElement}
            Else
```

```
   //If not, get the value from the source process
               GET PROCESS VARIABLE($lSrcProcess;$vpSrcObject->{$vlSrcElement};$vgDraggedPict)
               [aTable]aPicture:=$vgDraggedPict
            End if
      End case
End case
```

**Note:** For other examples showing how to handle On Drag Over and On Drop events, see the examples of the **DRAG AND DROP PROPERTIES** command.

## Example 4

This example is a template for a form method. It shows each of the possible events that can occur when a summary report uses a form as an output form:

```
   //Method of a form being used as output form for a summary report
$vpFormTable:=Current form table
Case of
   //...
   :(Form event=On Header)
   //A header area is about to be printed
      Case of
         :(Before selection($vpFormTable->))
   //Code for the first break header goes here
         :(Level=1)
   //Code for a break header level 1 goes here
         :(Level=2)
   //Code for a break header level 2 goes here
   //...
      End case
   :(Form event=On Printing Detail)
   //A record is about to be printed
   //Code for each record goes here
   :(Form event=On Printing Break)
   //A break area is about to be printed
      Case of
         :(Level=0)
   //Code for a break level 0 goes here
         :(Level=1)
   //Code for a break level 1 goes here
   //...
      End case
   :(Form event=On Printing Footer)
      If(End selection($vpFormTable->))
   //Code for the last footer goes here
      Else
   //Code for a footer goes here
      End if
End case
```

## Example 5

This example shows the template of a form method that handles the events that can occur for a form displayed using the **DISPLAY SELECTION** or **MODIFY SELECTION** commands. For didactic purposes, it displays the nature of the event in the title bar of the form window.

```
   //A form method
Case of
   :(Form event=On Load)
      $vsTheEvent:="The form is about to be displayed"
   :(Form event=On Unload)
      $vsTheEvent:="The output form has been exited and is about to disappear from the screen"
   :(Form event=On Display Detail)
      $vsTheEvent:="Displaying record #"+String(Selected record number([TheTable]))
   :(Form event=On Menu Selected)
      $vsTheEvent:="A menu item has been selected"
```

```
    :(Form event=On Header")
       $vsTheEvent:="The header area is about to be drawn"
    :(Form event=On Clicked")
       $vsTheEvent:="A record has been clicked"
    :(Form event=On Double Clicked")
       $vsTheEvent:="A record has been double clicked"
    :(Form event=On Open Detail)
       $vsTheEvent:="The record #"+String(Selected record number([TheTable]))+" is double-clicked"
    :(Form event=On Close Detail)
       $vsTheEvent:="Going back to the output form"
    :(Form event=On Activate)
       $vsTheEvent:="The form's window has just become the frontmost window"
    :(Form event=On Deactivate)
       $vsTheEvent:="The form's window is no longer the frontmost window"
    :(Form event=On Menu Selected)
       $vsTheEvent:="A menu item has been chosen"
    :(Form event=On Outside Call)
       $vsTheEvent:="A call from another has been received"
    Else
       $vsTheEvent:="What's going on? Event #"+String(Form event)
End case
SET WINDOW TITLE($vsTheEvent)
```

## Example 6

For examples on how to handle On Before Keystroke and On After Keystroke events, see examples for the **Get edited text**, **Keystroke** and **FILTER KEYSTROKE** commands.

## Example 7

This example shows how to treat clicks and double clicks in the same way in a scrollable area:

```
 //asChoices scrollable area object method
Case of
   :(Form event=On Load)
      ARRAY TEXT(asChoices;...)
 //...
      asChoices:=0
   :((Form event=On Clicked)|(Form event=On Double Clicked))
      If(asChoices#0)
 //An item has been clicked, do something here
 //...
      End if
 //...
End case
```

## Example 8

This example shows how to treat clicks and double clicks using a different response. Note the use of the element zero for keeping track of the selected element:

```
 //asChoices scrollable area object method
Case of
   :(Form event=On Load)
      ARRAY TEXT(asChoices;...)
 // ...
      asChoices:=0
      asChoices{0}:="0"
   :(Form event=On Clicked)
      If(asChoices#0)
         If(asChoices#Num(asChoices))
 //A new item has been clicked, do something here
 //...
```

```
   //Save the new selected element for the next time
            asChoices{0}:=String(asChoices)
         End if
      Else
         asChoices:=Num(asChoices{0})
      End if
   :(Form event=On Double Clicked)
      If(asChoices#0)
   //An item has been double clicked, do something different here
      End if
   // ...
End case
```

## Example 9

This example shows how to maintain a status text information area from within a form method, using the On Getting Focus and On Losing Focus events:

```
   //[Contacts];"Data Entry" form method
Case of
   :(Form event=On Load)
      C_TEXT(vtStatusArea)
      vtStatusArea:=""
   :(Form event=On Getting Focus)
      RESOLVE POINTER(Focus object;$vsVarName;$vlTableNum;$vlFieldNum)
      If(($vlTableNum#0) & ($vlFieldNum#0))
         Case of
            :($vlFieldNum=1) //Last name field
               vtStatusArea:="Enter the Last name of the Contact; it will be capitalized automatically"
   //...
            :($vlFieldNum=10) //Zip Code field
               vtStatusArea:="Enter a 5-digit zip code; it will be checked and validated automatically"
   //...
         End case
      End if
   :(Form event=On Losing Focus)
      vtStatusArea:=""
   //...
End case
```

## Example 10

This example shows how to respond to a close window event with a form used for record data entry:

```
   //Method for an input form
$vpFormTable:=Current form table
Case of
   //...
   :(Form event=On Close Box)
      If(Modified record($vpFormTable->))
         CONFIRM("This record has been modified. Save Changes?")
         If(OK=1)
            ACCEPT
         Else
            CANCEL
         End if
      Else
         CANCEL
      End if
   //...
End case
```

## Example 11

This example shows how to capitalize a text or alphanumeric field each time its data source value is modified:

```
 //[Contacts]First Name Object method
Case of
 //...
   :(Form event=On Data Change)
      [Contacts]First Name:=Uppercase(Substring([Contacts]First Name;1;1))+Lowercase(Substring([Contacts]First Name;2))
 //...
End case
```

## Example 12

This example shows how to capitalize a text or alphanumeric field each time its data source value is modified:

```
 //[Contacts]First Name Object method
Case of
 //...
   :(Form event=On Data Change)
      [Contacts]First Name:=Uppercase(Substring([Contacts]First Name;1;1))+Lowercase(Substring([Contacts]First Name;2))
 //...
End case
```

## Example 13

The following example illustrates how to manage a deletion action in a hierarchical list:

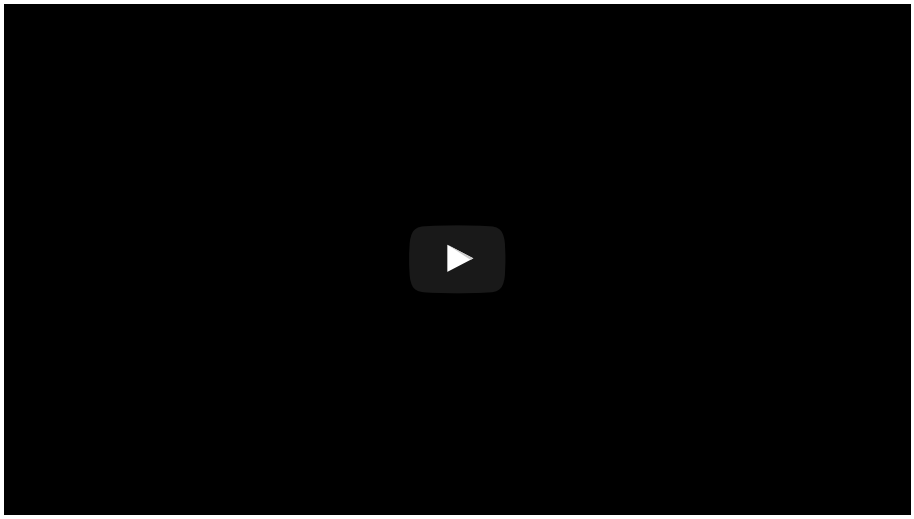
```
... //method of hierarchical list
:($Event=On Delete Action)
ARRAY LONGINT($itemsArray;0)
$Ref:=Selected list items(<>HL;$itemsArray;*)
$n:=Size of array($itemsArray)

Case of
   :($n=0)
      ALERT("No item selected")
      OK:=0
   :($n=1)
      CONFIRM("Do you want to delete this item?")
   :($n>1)
      CONFIRM("Do you want to delete these items?")
End case

If(OK=1)
   For($i;1;$n)
      DELETE FROM LIST(<>HL;$itemsArray{$i};*)
   End for
End if
```

## Example 14

In this example, the On Scroll form event allows us to synchronize the display of two pictures in a form. The following code is added in the "satellite" object method (picture field or variable):

```
Case of
   :(Form event=On Scroll)
 // we take the position of the left picture
      OBJECT GET SCROLL POSITION(*;"satellite";vPos;hPos)
 // and we apply it to the right picture
      OBJECT SET SCROLL POSITION(*;"plan";vPos;hPos;*)
End case
```

Result:



## Example 15

You want to draw a red rectangle around the selected cell of a list box, and you want the rectangle to move along with the list box if it is scrolled vertically by the user. In the list box object method, you can write:

```
Case of

   :(Form event=On Clicked)
      LISTBOX GET CELL POSITION(*;"LB1";$col;$raw)
      LISTBOX GET CELL COORDINATES(*;"LB1";$col;$raw;$x1;$y1;$x2;$y2)
      OBJECT SET VISIBLE(*;"RedRect";True) //initialize a red rectangle
      OBJECT SET COORDINATES(*;"RedRect";$x1;$y1;$x2;$y2)

   :(Form event=On Scroll)
      LISTBOX GET CELL POSITION(*;"LB1";$col;$raw)
      LISTBOX GET CELL COORDINATES(*;"LB1";$col;$raw;$x1;$y1;$x2;$y2)
      OBJECT GET COORDINATES(*;"LB1";$xlb1;$ylb1;$xlb2;$ylb2)
      $toAdd:=LISTBOX Get headers height(*;"LB1") //height of the header so as not to overlap it
      If($ylb1+$toAdd<$y1) & ($ylb2>$y2) //if we are inside the list box
 //to keep it simple, we only handle headers
 //but we should handle horizontal clipping
 //as well as scroll bars
         OBJECT SET VISIBLE(*;"RedRect";True)
         OBJECT SET COORDINATES(*;"RedRect";$x1;$y1;$x2;$y2)
      Else
         OBJECT SET VISIBLE(*;"RedRect";False)
      End if

End case
```

As a result, the red rectangle follows the scrolling of the list box:

| John | Mark | Amy | Jenny |
|---|---|---|---|
| 22072 | 30812 | 10426 | 24142 |
| 21858 | 17845 | 9899 | 23066 |
| 6773 | 12133 | 17423 | 21653 |
| 5269 | 32436 | 32124 | 24586 |
| 8555 | 32658 | 1868 | 9386 |
| 932 | 11022 | 19487 | 21255 |
| 26992 | 25056 | 31575 | 9882 |
| 771 | 14049 | 10139 | 30782 |
| 10520 | 18829 | 30037 | 24754 |
| 4969 | 12424 | 22836 | 27418 |

| John | Mark | Amy | Jenny |
|---|---|---|---|
| 5833 | 8131 | 31237 | 26638 |
| 26183 | 18940 | 21758 | 19336 |
| 17950 | 1912 | 7867 | 8335 |
| 21974 | 29957 | 25463 | 9780 |
| 9724 | 18580 | 12720 | 20457 |
| 16031 | 3003 | 10409 | 18439 |
| 13782 | 26164 | 5865 | 584 |
| 22072 | 30812 | 10426 | 24142 |
| 21858 | 17845 | 9899 | 23066 |
| 6773 | 12133 | 17423 | 21653 |

# In break

In break -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | Returns True if the execution cycle is in break |

## Description

**In break** returns True for the In break execution cycle.

In order for the **In break** execution cycle to be generated, make sure that the On Printing Break event property for the form and/or the objects has been selected in the Design environment.

**Note:** This command is equivalent to using **Form event** and testing whether it returns the On Printing Break event.

# In footer

In footer -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean |  | Returns True if the execution cycle is in footer |

## Description

**In footer** returns True for the In footer execution cycle.

In order for the **In footer** execution cycle to be generated, make sure that the On Printing Footer event property for the form and/or the objects has been selected in the Design environment.

**Note:** This command is equivalent to using **Form event** and testing whether it returns the On Printing Footer event.

# In header

In header -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | Returns True if the execution cycle is in header |

## Description

**In header** returns True for the In header execution cycle.

In order for the **In header** execution cycle to be generated, make sure that the On Header event property for the form and/or the objects has been selected in the Design environment.

**Note:** This command is equivalent to using **Form event** and testing whether it returns the On Header event.

# Is waiting mouse up

Is waiting mouse up -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | True if the object is waiting for a mouse up event, False otherwise |

## Description

The **Is waiting mouse up** command returns **True** if the current object has been clicked and the mouse button has not been released, while the parent window has the focus. Otherwise, it returns **False**, in particular if the parent window has lost the focus before the mouse button was released.

The command must be called in the context of the current object. It is designed to be used in conjunction with the On Mouse Up form event which is available for picture fields and variables. It allows your code to handle the cases where the user clicked and started to move something within a form object picture, and this action is interrupted by an external event, such as an alert dialog box. In this case, the internal state of the object can be suspended indefinitely because it is waiting for a mouse up event which will never occur. To prevent this issue, you must protect your mouse moving code within a **Is waiting mouse up** command which makes you sure that it is executed in a valid context.

## Example

The following code can be used to manage a mouse tracking feature in a picture object:

```
 //Object method of the picture object
C_LONGINT(vLtracking) //flag for tracking mode
Case of
   :(Form event=On Clicked)
      If(Is waiting mouse up) //the mouse button is still not released
         vLtracking:=1 //we are in tracking mode
 //... Write here initialization code for the tracking feature
      End if
   :(Form event=On Mouse Move)
      If(vLtracking=1) //we are in tracking mode
         If(Not(Is waiting mouse up)) //we'll never have the mouse up
            vLtracking:=0 //stop the tracking mode
 //... Write here the code for handling or canceling user tracking action
         Else //the object is still waiting for a mouse up
 //... Write here the code for the tracking
         End if
      End if
   :(Form event=On Mouse Up) //the mouse button was released
 //... Write here the code to complete the tracking action
      vLtracking:=0 //end of the tracking mode
End case
```

# Outside call

Outside call -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | True if the execution cycle is an outside call |

## Description

**Outside call** returns True for the After execution cycle.

In order for the **Outside call** execution cycle to be generated, make sure that the On Outside Call event property for the form and/or the objects has been selected in the Design environment.

**Note:** This command is equivalent to using **Form event** and testing whether it returns the On Outside Call event.

# Right click

Right click -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | True if a right click was detected, otherwise False |

## Description

The **Right click** command returns True if the right button of the mouse has been clicked.

This command should be used only in the context of the On Clicked form event. It is therefore necessary to verify in Design mode that the event has been properly selected in the Form properties and/or in the specific object.

# SET TIMER

SET TIMER ( tickCount )

| Parameter | Type | | Description |
|---|---|---|---|
| tickCount | Longint | ⇒ | Tickcount or -1=Trigger as soon as possible |

## Description

The **SET TIMER** command activates the On Timer form event and sets, for the current form and process, the number of ticks elapsed between each On Timer form event.

**Note:** For more information about this new form event, please refer to the description of the command **Form event**.

If this command is called in a context in which it is not displaying a form, it will have no effect.

**Note:** When the **SET TIMER** command is executed in the context of a subform (form method of the subform), the On Timer event is generated in the subform and not at the parent form level.

If you pass -1 in the *tickCount* parameter, the command will activate the On Timer form event "as soon as possible", in other words, as soon as the 4D application hands over control to the event manager. More particularly, this means that you can make sure that a form is completely displayed before beginning processing (application fluidity).

To procedurally disable the triggering of the On Timer form event, call **SET TIMER** again and pass 0 in *tickCount*.

## Example

Let's imagine that you want, when a form is displayed on screen, the computer to beep every three seconds. To do so, write the following form method:

```
If(Form event=On Load)
   SET TIMER(60*3)
End if

If(Form event=On Timer)
   BEEP
End if
```

# _o_During

_o_During -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | Returns True if the execution cycle is during |

## Compatibility note

Use of this old generic execution cycle function is not recommended. It is recommended to use **Form event** and test the specific events returned, like On Clicked.

# Forms

- CALL FORM
- Current form name
- FORM FIRST PAGE
- FORM Get current page
- FORM GET HORIZONTAL RESIZING
- FORM GET OBJECTS
- FORM GET PARAMETER
- FORM GET PROPERTIES
- FORM GET VERTICAL RESIZING
- FORM GOTO PAGE
- FORM LAST PAGE
- FORM LOAD
- FORM NEXT PAGE
- FORM PREVIOUS PAGE
- FORM SCREENSHOT
- FORM SET HORIZONTAL RESIZING
- FORM SET INPUT
- FORM SET OUTPUT
- FORM SET SIZE
- FORM SET VERTICAL RESIZING
- FORM UNLOAD

# CALL FORM

CALL FORM ( window ; method {; param}{; param2 ; ... ; paramN} )

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number |
| method | Text | ⇒ | Name of project method to call |
| param | Expression | ⇒ | Parameter(s) passed to method |

## Description

The **CALL FORM** command executes the project method whose name you passed in *method* with the optional *param*(s) in the context of a form displayed in a *window*, regardless of the process owning the window.

Just like in the worker-based interprocess communication feature (see **About workers**), a message box is associated with the window and can be used when the window displays a form (after the On Load form event). **CALL FORM** encapsulates the method name and its arguments in a message that is posted in the window's message box. The form then executes the message in its own process. The calling process can be cooperative or preemptive, thus this feature allows a preemptive process to exchange information with forms.

In *window*, you pass the window reference number of the window displaying the called form.

In *method*, you pass the name of the project method to be executed in the context of the *window* parent process.

You can also pass parameters to the method using one or more *param* parameters. You pass parameters the same way you would pass them to a subroutine (see the **Passing Parameters to Methods** section). Upon starting execution in the context of the form, the method receives the parameter values in *$1*, *$2*, and so on. Remember that arrays cannot be passed as parameters to a method. Furthermore, in the context of the **CALL FORM** command, the following additional considerations need to be taken into account:

- Pointers to tables or fields are allowed.
- Pointers to variables, particularly local and process variables, are not recommended since these variables may be undefined at the moment they are being accessed by the process method.
- If you pass an Object type parameter, 4D creates a copy of the object in the destination process if the form is in a process different from the one calling the **CALL FORM** command.

## Example 1

You want to open two different dialog windows from the same form, but with different background colors and different messages. You also want to send messages afterwards and display them in each dialog window.

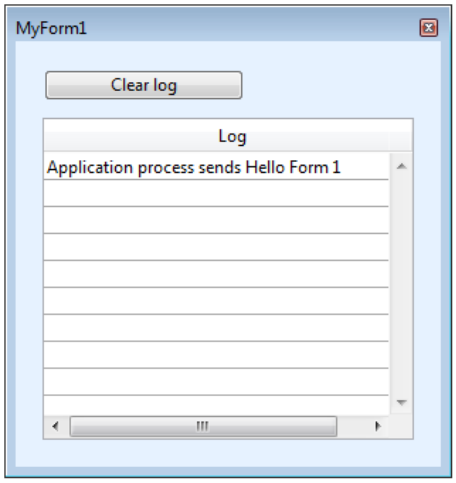
In the main form, a button opens the two dialogs:

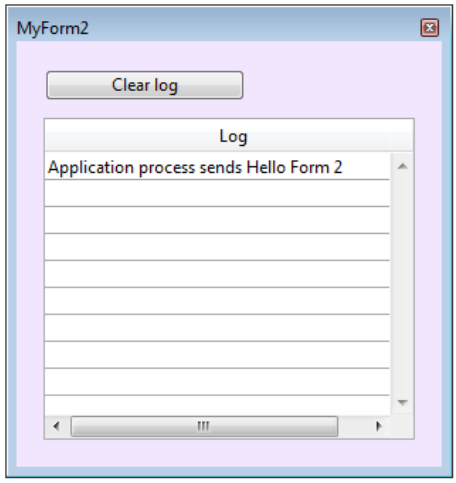
```
 //Object method to create forms
 //First window
formRef1:=Open form window("FormMessage";Palette form window;On the left)
SET WINDOW TITLE("MyForm1";formRef1)
DIALOG("FormMessage";*)
SHOW WINDOW(formRef1)

 //Second window
formRef2:=Open form window("FormMessage";Palette form window;On the left+500)
SET WINDOW TITLE("MyForm2";formRef2)
DIALOG("FormMessage";*)
SHOW WINDOW(formRef2)

 //Send colors
CALL FORM(formRef1;"doSetColor";0x00E6F2FF)
CALL FORM(formRef2;"doSetColor";0x00F2E6FF)
 //Create messages
CALL FORM(formRef1;"doAddMessage";Current process name;"Hello Form 1")
CALL FORM(formRef2;"doAddMessage";Current process name;"Hello Form 2")
```

The *doAddMessage* method only adds a row in the list box in the "FormMessage" form:

```
C_TEXT($1) //Caller name
C_TEXT($2) //Message to display
 //Receive message from $2 and log the message in the list box
$p:=OBJECT Get pointer(Object named;"Column1")
INSERT IN ARRAY($p->;1)
$p->{1}:=$1+" sends "+$2
```

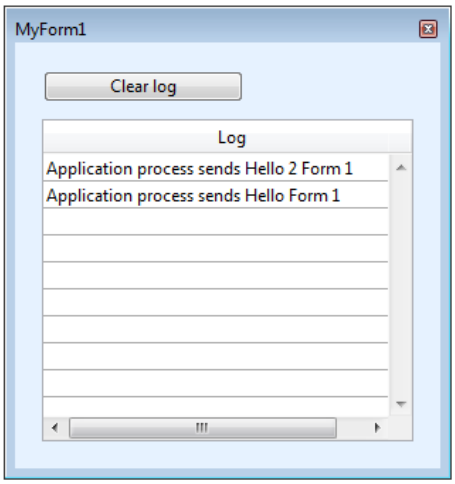
At runtime, you get the following result:

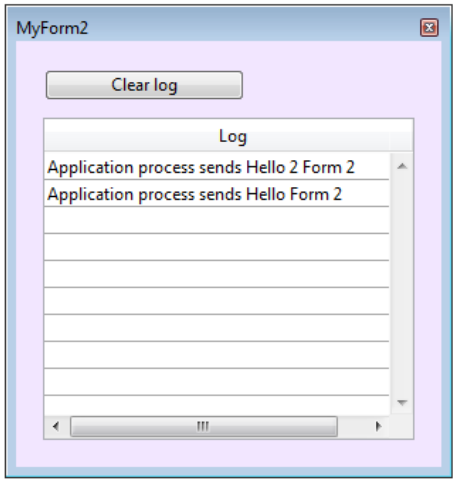



You can then add other messages by executing the **CALL FORM** command again:

```
CALL FORM(formRef1;"doAddMessage";Current process name;"Hello 2 Form 1")
CALL FORM(formRef2;"doAddMessage";Current process name;"Hello 2 Form 2")
```




## Example 2

You can use the **CALL FORM** command to pass custom settings to a form, for example configuration values, without having to use process variables:

```
$win:=Open form window("form")
CALL FORM($win;"configure";param1;param2)
DIALOG("form")
```

# Current form name

Current form name -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Text | ➲ | Name of current project form or current table form in the process |

## Description

The **Current form name** command returns the name of the current form defined for the process. The current form can be a project form or a table form.

By default, if you have not called the **FORM LOAD** command in the current process, the current form is the one being displayed or printed. If you have called the **FORM LOAD** command in the process, the current form is the one set by this command and it remains so until you call **FORM UNLOAD** (or **CLOSE PRINTING JOB**).

If there is no current form defined for the process, the command returns an empty string.

## Example 1

In an input form, place the following code in a button:

```
C_TEXT($FormName)
$win:=Open form window([Members];"Input";Plain form window)
DIALOG([Members];"Input")
$FormName:=Current form name
 // $FormName = "Input"
FORM LOAD([Members];"Drag")
$FormName:=Current form name
 // $FormName = "Drag"
 //...
```

## Example 2

You want to get the current form if it is a project form:

```
$PointerTable:=Current form table
If(Nil($PointerTable)) // this is a project form
   $FormName:=Current form name
   ... // processing
End if
```

# FORM FIRST PAGE

FORM FIRST PAGE
Does not require any parameters

## Description

**FORM FIRST PAGE** changes the currently displayed form page to the first form page. If no form is being displayed or loaded by the **FORM LOAD** command, or if the first form page is already displayed, **FORM FIRST PAGE** does nothing.

## Example

The following example is a one-line method called from a menu command. It displays the first form page:

```
FORM FIRST PAGE
```

# FORM Get current page

FORM Get current page {( * )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | Returns number of current subform page |
| Function result | Longint | ⇐ | Number of currently displayed form page |

## Description

The **FORM Get current page** command returns the number of the currently displayed form page or of the current form loaded by the **FORM LOAD** command.

The * parameter is useful when the command is called in the context of a page type subform containing several pages. In this case, when you pass this parameter, the command changes the page of the current subform (the one that called the command). By default, when the * parameter is omitted, the command is always applied to the parent form.

## Example

In a form, when you select a menu item from the menu bar or when the form receives a call from another process, you can perform different actions depending on the form page currently displayed. In this example, you write:

```
  ` [myTable];"myForm" Form Method
Case of
   :(Form event=On Load)
  ` ...
   :(Form event=On Unload)
  ` ...
   :(Form event=On Menu Selected)
      $vlMenuNumber:=Menu selected>>16
      $vlItemNumber:=Menu selected & 0xFFFF
      Case of
         :($vlMenuNumber=...)
            Case of
               :($vlItemNumber=...)
               :(FORM Get current page=1)
  ` Do appropriate action for page 1
               :(FORM Get current page=2)
  ` Do appropriate action for page 2
  ` ...
               :($vlItemNumber=...)
  ` ...
            End case
         :($vlMenuNumber=...)
  ` ...
      End case
   :(Form event=On Outside Call)
      Case of
         :(FORM Get current page=1)
  ` Do appropriate reply for page 1
         :(FORM Get current page=2)
  ` Do appropriate reply for page 2
      End case
  ` ...
End case
```

# FORM GET HORIZONTAL RESIZING

FORM GET HORIZONTAL RESIZING ( resize {; minWidth {; maxWidth}} )

| Parameter | Type | | Description |
|---|---|---|---|
| resize | Boolean | ⇦ | True: Form can be resized horizontally False: Form cannot be resized horizontally |
| minWidth | Longint | ⇦ | Smallest form width allowed (pixels) |
| maxWidth | Longint | ⇦ | Largest form width allowed (pixels) |

## Description

The **FORM GET HORIZONTAL RESIZING** command returns the horizontal resizing properties of the current form in the *resize*, *minWidth* and *maxWidth* variables. These properties may have been set for the form in the Form editor in Design mode or for the current process via the **FORM SET HORIZONTAL RESIZING** command.

# FORM GET OBJECTS

FORM GET OBJECTS ( objectsArray {; variablesArray {; pagesArray}} {; formPageOption | *} )

| Parameter | Type | | Description |
|---|---|---|---|
| objectsArray | String array | ⇦ | Name of form objects |
| variablesArray | Pointer array | ⇦ | Pointers to variables or fields associated with objects |
| pagesArray | Integer array | ⇦ | Page number of each object |
| formPageOption \| * | Longint, Operator | ⇨ | 1=Form current page, 2=Form all pages, 4=Form inherited<br>If * passed (obsolete) = current page with inherited objects |

## Description

The **FORM GET OBJECTS** command returns the list of all objects present in the current form in the form of (an) array(s). This list can be restricted to the current form page and can exclude objects of inherited forms. The command can be used with both input and output forms.

If an array passed as a parameter is not previously declared, the command creates it and automatically sets its size. However, in the interest of compiling the application, we recommend that you explicitly declare each array.

Pass the name of the string array that will contain object names (each object name is unique within a form) in *objectsArray*. The order in which objects appear in the array is not significant.

The other arrays optionally filled by the command are synchronized with the first array.

Pass the name of the pointer array that already contains pointers to variables or fields associated with objects in the optional *variablesArray* parameter. If an object does not have an associated variable, the pointer **Nil** is returned. If there is a “subform” type object, a pointer to the table of the subform is returned.

The third array (optional), *pagesArray*, is filled with the form page numbers. Each line of this array contains the page number of the corresponding object.

The optional * parameter allows you to reduce the list of objects returned to the current page of the form. When this parameter is passed, only objects of the current page, page 0 and inherited pages are returned by the command. In other words, all the objects present in the current page of the form (visible or not) are processed by the command.

The optional *formPageOption* parameter allows you to specify the form part(s) from where you want to get objects. By default, if the *formPageOption* parameter is omitted (as well as the * parameter), objects from all pages, including inherited objects, are returned. To reduce the scope of the command, you can pass a value in *formPageOption*. You can pass one (or a combination) of the following constants, found in the "**Form Objects (Access)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Form all pages | Longint | 2 | Returns all objects of all the pages, excluding inherited objects |
| Form current page | Longint | 1 | Returns all objects of the current page, including page 0 but excluding inherited objects |
| Form inherited | Longint | 4 | Returns inherited objects only |

**Compatibility note:** Passing the * parameter is equivalent to passing Form current page+Form inherited. The syntax using the * parameter is now deprecated and should no longer be used.

## Example 1

You want to get information on all pages including objects from the inherited form (if any):

```
 //open form
FORM GET OBJECTS(objectsArray;variablesArray;pagesArray)
```

Or:

```
 //loaded form
FORM LOAD([Table1];"MyForm")
FORM GET OBJECTS(objectsArray;variablesArray;pagesArray;Form all pages+Form inherited)
```

## Example 2

You want to get information on the current page only, with page 0 of the loaded form and inherited form objects (if any):

```
FORM LOAD("MyForm")
FORM GOTO PAGE(2)
FORM GET OBJECTS(objectsArray;variablesArray;pagesArray;Form current page+Form inherited)
```

## Example 3

You want to get information on all objects in the inherited form (if any). If there is no inherited form, arrays will be returned empty.

```
FORM LOAD("MyForm")
FORM GET OBJECTS(objectsArray;variablesArray;pagesArray;Form inherited)
```

## Example 4

You want to get information on page 4 objects, including page 0 objects, but without inherited form objects (if any):

```
FORM LOAD([Table1];"MyForm")
FORM GOTO PAGE(4)
FORM GET OBJECTS(objectsArray;variablesArray;pagesArray;Form current page)
```

## Example 5

You want to get information on objects on all pages, but without inherited form objects (if any):

```
FORM LOAD([Table1];"MyForm")
FORM GET OBJECTS(objectsArray;variablesArray;pagesArray;Form all pages)
```

# FORM GET PARAMETER

FORM GET PARAMETER ( {aTable ;} form ; selector ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Form table or Default table if this parameter is omitted |
| form | String | ⇒ | Form name |
| selector | Longint | ⇒ | Code of the parameter |
| value | Longint | ⇐ | Current value of the parameter |

## Description

The **FORM GET PARAMETER** command can be used to get the current *value* of a parameter of the form indicated by *aTable* and *form*.

*selector* indicates the parameter of the form whose value you want to find out. You can use the following constant, found in the "**Form Parameters**" theme:

| Constant | Type | Value |
|---|---|---|
| NonInverted objects | Longint | 0 |

When you use the NonInverted Objects constant as *selector*, the command returns, in *value*, the actual display mode of the form in Application mode under Windows. This parameter is used when applications are displayed using "right-to-left" languages. For more information about the support of right-to-left languages, please refer to the *Design Reference* manual of 4D.

- If *value* returns 0, the form objects are inverted,
- If *value* returns 1, the form objects are not inverted.

If the command is not called within the context of the Application mode under Windows, it always returns 1.

Keep in mind that the actual inversion of form objects will depend on a combination of several parameters: the values of the "Inversion of objects in Application mode" preference, the value of the "Do not invert objects" form option and the system on which the database is running. The following table specifies the value returned by the **FORM GET PARAMETER** command depending on the various combinations of these parameters:

| Preferences: "Inversion of objects in Application mode" (1) | Form property: "Do not invert objects" | Right-to-left Display under Windows | Value returned by FORM GET PARAMETER |
|---|---|---|---|
| No | X | X | 1 |
| | | X | 1 |
| | X | | 1 |
| | | | 1 |
| Automatic | X | X | 1 |
| | | X | 0 |
| | X | | 1 |
| | | | 1 |
| Yes | X | X | 1 |
| | | X | 0 |
| | X | | 1 |
| | | | 0 |

(1) This preference can also be set or read using the **SET DATABASE PARAMETER** and **Get database parameter** commands.

# FORM GET PROPERTIES

FORM GET PROPERTIES ( {aTable ;} formName ; width ; height {; numPages {; fixedWidth {; fixedHeight {; title}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table of the form or Default table, if omitted |
| formName | String | ⇒ | Name of the form |
| width | Longint | ⇐ | Width of the form (in pixels) |
| height | Longint | ⇐ | Height of the form (in pixels) |
| numPages | Longint | ⇐ | Number of pages in the form |
| fixedWidth | Boolean | ⇐ | True = Fixed width, False = Variable width |
| fixedHeight | Boolean | ⇐ | True = Fixed height, False = Variable height |
| title | Text | ⇐ | Title of the form's window |

## Description

The **FORM GET PROPERTIES** command returns the properties of the form *formName*.

The *width* and *height* parameters return the form's width and height in pixels. These values are determined from the form's Default window size properties:

- If the form's size is **automatic**, its width and height are calculated so that all the form's objects are visible, by taking into consideration the horizontal and vertical margins that were defined.
- If the form's size is **set**, its width and height are those manually entered in the corresponding areas.
- If the form's size is **based on an object**, its width and height are calculated in relation to this object's position.

The *numPages* parameter returns the number of pages in the form, excluding page 0 (zero).

The *fixedWidth* and *fixedHeight* parameters indicate if the length and width of the form are resizable (the parameter returns **False**) or set (the parameter returns **True**).

The *title* parameter returns the title of the form's window as it was defined in the Property List of the Form editor. If no name was defined, the *title* parameter returns an empty string.

# FORM GET VERTICAL RESIZING

FORM GET VERTICAL RESIZING ( resize {; minHeight {; maxHeight}} )

| Parameter | Type | | Description |
|---|---|---|---|
| resize | Boolean | ⇦ | True: Form can be resized vertically False: Form cannot be resized vertically |
| minHeight | Longint | ⇦ | Smallest form height allowed (pixels) |
| maxHeight | Longint | ⇦ | Largest form height allowed (pixels) |

## Description

The **FORM GET VERTICAL RESIZING** command returns the vertical resizing properties of the current form in the *resize*, *minHeight* and *maxHeight* variables. These properties may have been set for the form in the Form editor in Design mode or for the current process via the **FORM SET VERTICAL RESIZING** command.

# FORM GOTO PAGE

FORM GOTO PAGE ( pageNumber {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| pageNumber | Longint | ⇒ | Form page to display |
| * | Operator | ⇒ | Change page of current subform |

## Description

**FORM GOTO PAGE** changes the currently displayed form page to the form page specified by *pageNumber*.

If no form is displayed or loaded by the **FORM LOAD** command, or if *pageNumber* corresponds to the current page of the form, **FORM GOTO PAGE** does nothing. If *pageNumber* is greater than the number of pages, the last page is displayed. If *pageNumber* is less than one, the first page is displayed.

The * parameter is useful when the command is called in the context of a page type subform containing several pages. In this case, when you pass this parameter, the command changes the page of the current subform (the one that called the command). By default, when the * parameter is omitted, the command is always applied to the parent form.

### About form page management commands

Automatic action buttons perform the same tasks as the **FORM FIRST PAGE**, **FORM LAST PAGE**, **FORM NEXT PAGE**, **FORM PREVIOUS PAGE** and **FORM GOTO PAGE** commands that you can apply to objects such as tab controls, drop-down list boxes, and so on. Whenever appropriate, use automatic action buttons instead of commands.

Page commands can be used with input forms or with forms displayed in dialogs. Output forms use only the first page. A form always has at least one page—the first page. Remember that regardless of the number of pages a form has, only one form method exists for each form.

- Use the **FORM Get current page** command to find out which page is being displayed.
- Use the On Page Change **Form event** that is generated each time the current page of the form changes.

**Note:** When **designing** a form, you can work with pages 1 through X, as well as with page 0, in which you put objects that will appear in all of the pages. When **using** a form, and therefore when calling page commands, you work with pages 1 through X; page 0 is automatically combined with the page being displayed.

## Example

The following example is an object method for a button. It displays a specific page, page 3:

```
FORM GOTO PAGE(3)
```

# FORM LAST PAGE

FORM LAST PAGE
Does not require any parameters

## Description

**FORM LAST PAGE** changes the currently displayed form page to the last form page. If a form is not being displayed or loaded by the **FORM LOAD** command, or if the last form page is already displayed, **FORM LAST PAGE** does nothing.

## Example

The following example is a one-line method called from a menu command. It displays the last form page:

```
FORM LAST PAGE
```

# FORM LOAD

FORM LOAD ( {aTable ;} form {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table form to load (if omitted, load a project form) |
| form | String | ⇒ | Name of form (project or table) to open |
| * | Operator | ⇒ | If passed = command applies to host database when it is executed from a component (parameter ignored outside of this context) |

## Description

The **FORM LOAD** command is used to load the *form* (project or table) in memory in the current process in order to print its data or parse its contents. There can only be one current form per process.

### Printing data

In order to be able to execute this command, a print job must be opened beforehand using the **OPEN PRINTING JOB** command. The **OPEN PRINTING JOB** command makes an implicit call to the **FORM UNLOAD** command, so in this context it is necessary to execute **FORM LOAD**. Once loaded, this *form* becomes the current printing form. All the object management commands, and in particular the **Print object** command, work with this form.

If a printing form has already been loaded beforehand (via a previous call to the **FORM LOAD** command), it is closed and replaced by *form*. You can open and close several project forms in the same print session. Changing the printing form via the **FORM LOAD** command does not generate page breaks. It is up to the developer to manage page breaks.

Only the On Load form event is executed during the opening of the project form, as well as any object methods of the form. Other form events are ignored. The On Unload form event is executed at the end of printing.

To preserve the graphic consistency of forms, it is recommended to apply the "Printing" appearance property regardless of the platform.

The current printing form is automatically closed when the **CLOSE PRINTING JOB** command is called.

**Compatibility note:** In versions of 4D prior to v14, the **FORM LOAD** command (named OPEN PRINTING FORM) accepted an empty string in the *form* parameter to close the current project form. This syntax is no longer supported and returns an error. You must use the **FORM UNLOAD** command or the **CLOSE PRINTING JOB** command in order to close the form.

### Parsing form contents

This consists in loading an off-screen form for parsing purposes. To do this, just call **FORM LOAD** outside the context of a print job. In this case, form events are not executed.

**FORM LOAD** can be used with the **FORM GET OBJECTS** and **OBJECT Get type** commands in order to perform any type of processing on the form contents. You must then call the **FORM UNLOAD** command in order to release the form from memory.

Note that in all cases, the form on screen remains loaded (it is not affected by the **FORM LOAD** command) so it is not necessary to reload it after calling **FORM UNLOAD**.

When the command is executed from a component, it loads the component forms by default. If you pass the * parameter, the method loads the host database forms.

**Reminder:** In the off-screen context, do not forget to call **FORM UNLOAD** to avoid any risk of memory overflow.

## Example 1

Calling a project form in a print job:

```
OPEN PRINTING JOB
FORM LOAD("print_form")
 // execution of events and object methods
```

## Example 2

Calling a table form in a print job:

```
OPEN PRINTING JOB
FORM LOAD([People];"print_form")
 // execution of events and object methods
```

## Example 3

Parsing of form contents to carry out processing on text input areas:

```
FORM LOAD([People];"my_form")
 // selection of form without execution of events or methods
FORM GET OBJECTS(arrObjNames;arrObjPtrs;arrPages;*)
For($i;1;Size of array(arrObjNames))
   If(OBJECT Get type(*;arrObjNames{$i})=Object type text input)
 //… processing
   End if
End for
FORM UNLOAD //do not forget to unload the form
```

# FORM NEXT PAGE

FORM NEXT PAGE
Does not require any parameters

## Description

**FORM NEXT PAGE** changes the currently displayed form page to the next form page. If no form is being displayed or loaded by the **FORM LOAD** command, or if the last form page is already displayed, **FORM NEXT PAGE** does nothing.

## Example

The following example is a one-line method called from a menu command. It displays the form page that follows the one currently displayed:

```
FORM NEXT PAGE
```

# FORM PREVIOUS PAGE

FORM PREVIOUS PAGE
Does not require any parameters

## Description

**FORM PREVIOUS PAGE** changes the currently displayed form page to the previous form page. If no form is being displayed or loaded by the **FORM LOAD** command, or if the first form page is already displayed, **FORM PREVIOUS PAGE** does nothing.

## Example

The following example is a one-line method called from a menu command. It displays the form page that precedes the one currently displayed:

```
FORM PREVIOUS PAGE
```

# FORM SCREENSHOT

FORM SCREENSHOT ( {{aTable ;} formName ;} formPict {; pageNum} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Form table |
| formName | Text | ⇒ | Name of form |
| formPict | Picture | ⇐ | Picture of form being executed if first parameter(s) omitted, or<br>Picture of form in Form editor if a form name is passed |
| pageNum | Longint | ⇒ | Form page number |

## Description

The **FORM SCREENSHOT** command returns a form as a picture. This command accepts two different syntaxes: depending on the syntax used, you get either a picture of an executed form, or a picture of the form in the Form editor.

- **FORM SCREENSHOT** ( *formPict* )
  This syntax gets the exact screenshot of the current page of the form being executed or loaded by means of the **FORM LOAD** command: the picture returned in the *formPict* parameter contains all the form's visible objects with the current field and variable values of the form, subform, and so on. The form is returned in full, without taking the size of the window that contains it into account.
  Note that this syntax only works with input forms.

- **FORM SCREENSHOT** ( {*aTable* ;} *formName* ; *formPict* {; *pageNum*} )
  This syntax gets a screenshot of a form "template" as it is displayed in the Form editor. All visible objects are drawn like they are in the editor; the command takes inherited forms and objects placed on page 0 into account.
  If you want a screenshot for a table form, pass the form table in the *aTable* parameter and then pass its name as a string in *formName*. For a project form, pass the form name directly in *formName*.
  By default, the command returns a screenshot of page 1 of the form. If you only want a picture of page 0, or any other page of the form, pass the desired page number in the *pageNum* parameter.

**Note:** Since the first two parameters of this command are optional, you cannot pass a function returning a pointer, such as **Current form table**-> or **Table**->, directly as an argument. Although this syntax would work in interpreted mode, it would be rejected during compilation, so instead you need to use an intermediate pointer variable in this case. For more information, refer to "**Direct use of commands returning pointers**".

# FORM SET HORIZONTAL RESIZING

FORM SET HORIZONTAL RESIZING ( resize {; minWidth {; maxWidth}} )

| Parameter | Type | | Description |
|---|---|---|---|
| resize | Boolean | ⇒ | True: The form can be resized horizontally False: The form cannot be resized horizontally |
| minWidth | Longint | ⇒ | Smallest form width allowed (pixels) |
| maxWidth | Longint | ⇒ | Largest form width allowed (pixels) |

## Description

The **FORM SET HORIZONTAL RESIZING** command allows you to change the horizontal resizing properties of the current form through programming. By default, these properties are set in the Design environment Form editor. New properties are set for the current process; they are not stored with the form.

The *resize* parameter lets you set whether the form can be resized horizontally; in other words, if the width can be changed (manually by the user or through programming).

If you pass **True**, the form width can be modified by the user; 4D uses values passed in *minWidth* and *maxWidth* as markers.

If you pass **False**, the current form width cannot be changed; in this case, there is no need to pass values in the *minWidth* and *maxWidth* parameters.

If you passed **True** in the first parameter, you can pass new minimum and maximum widths (in pixels) in the optional *minWidth* and *maxWidth* parameters. If you leave these parameters out, the values set in the Design environment (if any) are used.

## Example

Refer to the example of the **FORM SET SIZE** command.

# FORM SET INPUT

FORM SET INPUT ( {aTable ;} form {; userForm}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to set the input form, or Default table, if omitted |
| form | String | ⇒ | Name of the form to set as input form |
| userForm | String | ⇒ | Name of user form to use |
| * | | ⇒ | Automatic window size |

## Description

The **FORM SET INPUT** command sets the current input form for *aTable* to *form* or *userForm*. The form must belong to *aTable*.

The scope of this command is the current process. Each table has its own input form in each process.

**Note:** For structural reasons, this command is not compatible with project forms. If you pass a project form in *form*, the command does nothing.

**FORM SET INPUT** does not display the form; it just designates which form is used for data entry, import, or operation by another command. For information about creating forms, see the 4D *Design Reference* manual.

The default input form is defined in the Explorer window for each table. This default input form is used if the **FORM SET INPUT** command is not used to specify an input form, or if you specify a form that does not exist.

The optional *userForm* parameter lets you specify a user form (coming from *form*) as the default input form. If you pass a valid user form name, this form will be used by default instead of the input form in the current process. This allows you to have several different custom user forms simultaneously (generated using the **CREATE USER FORM** command) and to use the one that suits according to the context.
For more information about user forms, refer to the **Overview of user forms** section.

Input forms are displayed by a number of commands, which are generally used to allow the user to enter new data or modify old data. The following commands display an input form for data entry or query purposes:

- **ADD RECORD**
- **DISPLAY RECORD**
- **MODIFY RECORD**
- **QUERY BY EXAMPLE**

The **DISPLAY SELECTION** and **MODIFY SELECTION** commands display a list of records using the output form. The user can double-click on a record in the list, which displays the input form.

The import commands **IMPORT TEXT**, **IMPORT SYLK** and **IMPORT DIF** use the current input form for importing records.

The optional * parameter is used in conjunction with the form properties you set in the Design environment Form Properties window and the command **Open window**. Specifying the * parameter tells 4D to use the form properties to automatically resize the window for the next use of the form (as an input form or as a dialog box). See **Open window** for more information.

**Note:** Whether or not you pass the optional * parameter, **FORM SET INPUT** changes the input form for the table.

## Example 1

The following example shows a typical use of **FORM SET INPUT**:

```
FORM SET INPUT([Companies];"New Comp") ` Form for adding new companies
ADD RECORD([Companies]) ` Add a new company
```

## Example 2

In an invoicing database managing several companies, the creation of an invoice must be carried out using the corresponding user form:

```
Case of
   :(company="4D SAS")
      FORM SET INPUT([Invoices];"Input";"4D_SAS")
   :(company="4D Inc")
      FORM SET INPUT([Invoices];"Input";"4D_Inc")
   :(company="Acme")
      FORM SET INPUT([Invoices];"Input";"ACME")
End case
ADD RECORD([Factures])
```

# FORM SET OUTPUT

FORM SET OUTPUT ( {aTable ;} form {; userForm} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to set the output form, or Default table, if omitted |
| form | String | ⇒ | Form name |
| userForm | String | ⇒ | Name of user form to use |

## Description

The **FORM SET OUTPUT** command sets the current output form for *table* to *form* or *userForm*. The form must belong to *aTable*.

The scope of this command is the current process. Each table has its own output form in each process.

**Note:** For structural reasons, this command is not compatible with project forms. If you pass a project form in *form*, the command does nothing.

**FORM SET OUTPUT** does not display the form; it just designates which form is printed, displayed, or used by another command. For information about creating forms, see the 4D *Design Reference* manual.

The default output form is defined in the Explorer window for each table. This default output form is used if the **FORM SET OUTPUT** command is not used to specify an output form, or if you specify a form that does not exist.

The optional *userForm* parameter lets you specify a user form (coming from *form*) as the default output form. If you pass a valid user form name, this form will be used by default instead of the output form in the current process. This allows you to have several different custom user forms simultaneously (generated using the **CREATE USER FORM** command) and to use the one that suits according to the context.

For more information about user forms, refer to the **Overview of user forms** section.

Output forms are used by three groups of commands. One group displays a list of records on screen, another group generates reports, and the third group exports data. The **DISPLAY SELECTION** and **MODIFY SELECTION** commands display a list of records using an output form. You use the output form when creating reports with the **PRINT LABEL** and **PRINT SELECTION** commands. Each of the export commands (**EXPORT DIF**, **EXPORT SYLK** and **EXPORT TEXT**) also uses the output form.

## Example

The following example shows a typical use of **FORM SET OUTPUT**. Note that although the **FORM SET OUTPUT** command appears immediately before the output form is used, this is not required. In fact, the command may be executed in a completely different method, as long as it is executed prior to this method:

```
FORM SET INPUT([Parts];"Parts In") ` Select the input form
FORM SET OUTPUT([Parts];"Parts List") ` Select the output form
MODIFY SELECTION([Parts]) ` This command uses both forms
```

# FORM SET SIZE

FORM SET SIZE ( {object ;} horizontal ; vertical {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| object | String | ⇒ | Object name indicating form limits |
| horizontal | Longint | ⇒ | If * passed: horizontal margin (pixels)<br>If * omitted: width (pixels) |
| vertical | Longint | ⇒ | If * passed: vertical margin (pixels)<br>If * omitted: height (pixels) |
| * | Operator | ⇒ | • If passed, use horizontal and vertical as form margins<br>• If omitted, use horizontal and vertical as width and height of the form<br>This parameter cannot be passed if the object parameter is passed. |

## Description

The **FORM SET SIZE** command allows you to change the size of the current form by programming. The new size is defined for the current process; it is not saved with the form.

As in the Design environment, you can use this command to set the form size in three ways:

- Automatically — 4D determines the size of the form based on the notion that all objects must be visible — and possibly adding a horizontal and vertical margin,
- On the place where a form object is found, where a horizontal and vertical margin may be added,
- By entering “fixed” sizes (width and height).
  For more information on resizing forms, refer to the 4D *Design Reference* manual.

**Automatic size**

If you want the size of the form to be set automatically, you must use the following syntax:

```
FORM SET SIZE(horizontal;vertical;*)
```

In this case, you must pass the margins (in pixels) that you want to add to the right and bottom of the form in *horizontal* and *vertical*.

**Object-based size**

If you want the form size to be based on an object, you must use the following syntax:

```
FORM SET SIZE(object;horizontal;vertical)
```

In this case, you must pass the margins (in pixels) that you want to add to the right and bottom of the object in *horizontal* and *vertical*. You cannot pass the * parameter.

**Fixed size**

In you want to have a fixed form size, you must use the following syntax:

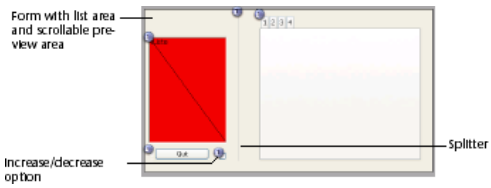
```
FORM SET SIZE(horizontal;vertical)
```

In this case, you must pass the width and height (in pixels) of the form in *horizontal* and *vertical*.

The **FORM SET SIZE** command changes the size of the form, but also takes into account the resizing properties. For example, if the minimum width of a form is 500 pixels and if the command sets a width of 400 pixels, the new form width will be 500 pixels.
Also note that this command does not change the size of the form window (you can resize a form without changing the size of the window and vice versa). To change the size of the form window, refer to the **RESIZE FORM WINDOW** command.

## Example

The following example shows how an Explorer type window is set up. The following form is created in the Design environment :

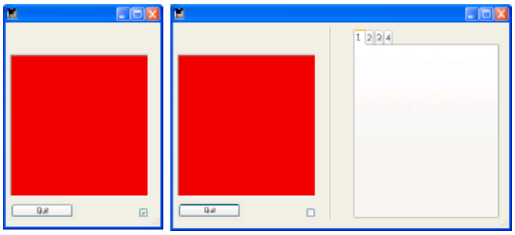



The size of the form is “automatic”.

The window is displayed using the following code:

```
$ref:=Open form window([Table 1];"Form1";Plain form window;Horizontally centered;Vertically centered;*)
DIALOG([Table 1];"Form1")
CLOSE WINDOW
```

The right part of the window can be displayed or hidden by clicking on the increase/decrease option:



The object method associated with this button is as follows:

```
Case of
   :(Form event=On Load)
      C_BOOLEAN(b1;<>collapsed)
      C_LONGINT(margin)
      margin:=15
      b1:=<>collapsed
      If(<>collapsed)
         FORM SET HORIZONTAL RESIZING(False)
         FORM SET SIZE("b1";margin;margin)
      Else
         FORM SET HORIZONTAL RESIZING(True)
         FORM SET SIZE("tab";margin;margin)
      End if

   :(Form event=On click)
      <>collapsed:=b1
      If(b1)
 `collapsed
         OBJECT GET COORDINATES(*;"b1";$l;$t;$r;$b)
         GET WINDOW RECT($lf;$tf;$rf;$bf;Current form window)
         SET WINDOW RECT($lf;$tf;$lf+$r+margin;$tf+$b+margin;Current form window)
         SET FORM HORIZONTAL RESIZING(False)
         SET FORM SIZE("b1";margin;margin)

      Else
 `expanded
         OBJECT GET COORDINATES(*;"tab";$l;$t;$r;$b)
         GET WINDOW RECT($lf;$tf;$rf;$bf;Current form window)
         SET WINDOW RECT($lf;$tf;$lf+$r+margin;$tf+$b+margin;Current form window)
         FORM SET HORIZONTAL RESIZING(True)
         FORM SET SIZE("tab";margin;margin)
      End if

End case
```

# FORM SET VERTICAL RESIZING

FORM SET VERTICAL RESIZING ( resize {; minHeight {; maxHeight}} )

| Parameter | Type | | Description |
|---|---|---|---|
| resize | Boolean | ⇒ | True: The form can be resized vertically<br>False: The form cannot be resized vertically |
| minHeight | Longint | ⇒ | Smallest form height allowed (pixels) |
| maxHeight | Longint | ⇒ | Largest form height allowed (pixels) |

## Description

The **FORM SET VERTICAL RESIZING** command allows you to change the vertical resizing properties of the current form through programming. By default, these properties are set in the Design environment Form editor. New properties are set for the current process; they are not stored with the form.

The *resize* parameter lets you set whether the form can be resized vertically; in other words, if the height can be changed (manually by the user or through programming).

If you pass **True**, the form height can be modified by the user; 4D uses values passed in *minHeight* and *maxHeight* as markers.

If you pass **False**, the current form height cannot be changed; in this case, there is no need to pass values in the *minHeight* and *maxHeight* parameters.

If you passed **True** in the first parameter, you can pass new minimum and maximum heights (in pixels) in the optional *minHeight* and *maxHeight* parameters. If you leave these parameters out, the values set in the Design environment (if any) are used.

## Example

Refer to the example of the **FORM SET SIZE** command.

# FORM UNLOAD

FORM UNLOAD
Does not require any parameters

## Description

The **FORM UNLOAD** command releases from memory the current form designated using the **FORM LOAD** command.

Calling this command is necessary when you use the **FORM LOAD** command outside of the printing context (in the case of printing, the current form is automatically closed again when the **CLOSE PRINTING JOB** command is called).

# Formulas

- Using tokens in formulas
- EDIT FORMULA
- EXECUTE FORMULA
- GET ALLOWED METHODS
- SET ALLOWED METHODS

# Using tokens in formulas

## Overview

4D's language includes a unique *tokenization* system for all object names of the language that are used in the code (constants, commands, tables, fields and keywords). *Tokenizing* these names means that as you type in the code editor they are stored internally as absolute references (numbers) and then restored as text during execution or display depending on the context. This allows you to guarantee that the code will always be interpreted correctly, even if you rename your tables or fields, or when 4D language commands are renamed over the course of different application versions.

**Note:** This also ensures automatic translation of the code when you have enabled the "Use regional system settings" option on the **Methods Page** of the Preferences and open your databases with 4D versions in different languages.

*Tokenisation* is completely transparent for 4D developers when working in the code editor. However, this mechanism is not automatically implemented in 4D formulas since they consist of text that is interpreted at runtime, and not as it is typed. In fact, this is the case as soon as 4D code is expressed as raw text, more specifically when code is exported and then imported using the **METHOD GET CODE** and **METHOD SET CODE** commands, copied/pasted or interpreted from **4D Transformation Tags**.
To continue to benefit from *tokenization* mechanisms in these contexts, you just need to use an explicit syntax (described below) which consists in preceding object names in the language by their *token*.

## Token syntax

By default, the *token* mechanism is not implemented automatically in 4D formulas (as well as contexts where 4D code is expressed as raw text, see above). As a result, for named elements contains in expressions, 4D offers a special syntax you can use to reference the *tokens* directly: you just need to add a specific suffix after the element name to indicate its type (command, field, etc.), followed by its reference. The **token syntax** is detailed in this table:

| Element | Example (standard syntax) | Suffix | Example (token syntax) | Comments |
|---|---|---|---|---|
| 4D Command | String(a) | :Cxx | String:C10(a) | xx is the command number |
| Table | [Employees] | :xx | [Employees:1] | xx is the table number |
| Field | [Employees]Name | :xx | [Employees:1]Name:2 | xx is the field number |
| 4D Plugin | PV PRINT(area) | :Pxx:yy | PV PRINT:P13000:229(area) | xx is the plug-in ID and yy is the index of the command |

**Note:** Uppercase letters (C, P) must be used in the suffixes; otherwise, they will not be interpreted correctly.

When you use this syntax, you guarantee that your formulas will be interpreted correctly even in the case of renaming or when the database is executed in a different language.

**Note :** Constants are also tokenized in the language however in formulas you can just pass their value in order make them independent of the context.

This syntax is accepted in all 4D formulas (or 4D expressions) regardless of the calling context:

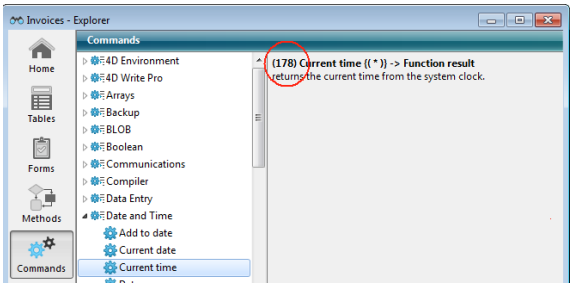
- 4D formulas executed using the **Formula editor** or using commands such as **EXECUTE FORMULA**, **APPLY TO SELECTION**, **QUERY BY FORMULA**, **LISTBOX INSERT COLUMN FORMULA**, etc.
- expressions inserted in rich text areas (see **ST INSERT EXPRESSION** and **Supported tags**),
- expressions calculated in transformation tags (see **4D Transformation Tags**),
- expressions inserted in plug-in areas,
- expressions inserted in 4D Write Pro areas (starting with 4D v15).

## Where to find the element numbers?

The token syntax requires the addition of the reference numbers of various elements. The location of these references depends on the type of element.

### 4D commands

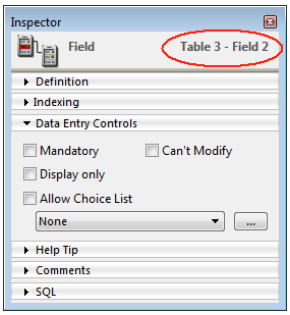
Command numbers can be found in this *Language Reference* manual ("Properties" area) as well as on the **Commands** page of the Explorer:




## Tables and fields

Table and field numbers can be obtained using the **Table** and **Field** commands.

They are also displayed in the **Inspector palette** of the Structure editor:




## 4D plug-in commands

To know what the tokens are for 4D plug-in commands, the trick is to enter the desired code in the Method editor and then restart 4D after disabling the plug-in (for example, by moving its folder). This means that only tokens will be displayed in the Method editor, and you can then copy the ones you need.

Code with plug-in installed:

```
12    PV SET ROWS HEIGHT (Area;1;10;PV Get row height (Area;10)+$Height)
```

The same code after plug-in has been disabled:

```
12    ô13000;75ô (Area;1;10;ô13000;76ô (Area;10)+$Height)
```

# EDIT FORMULA

EDIT FORMULA ( aTable ; formula )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to display by default in the Formula editor |
| formula | String variable | ⇒ | Variable containing the formula to display in the Formula editor or "" to display editor only |
| | | ⇐ | Formula validated by the user |

## Description

The **EDIT FORMULA** command displays the Formula editor in order to let the user write or modify a formula. The editor contains the following on opening:

- in the left list, the fields of the table passed in the *table* parameter,
- in the formula area, the formula contained in the *formula* variable. If you passed an empty string in *formula*, the Formula editor is displayed without a formula.

The user can modify the *formula* displayed and save it. It is also possible to write or load a new formula. Regardless, if the user validates the dialog box, the system variable OK is set to 1 and the *formula* variable contains the formula defined by the user. If the user cancels the formula, the system variable OK is set to 0 and the *formula* variable is left untouched.

**Notes:**

- By default, access to methods and commands is restricted for all users (except for the Designer and Administrator, in databases created with 4D 2004.4 and higher). When this mechanism is enabled, you must explicitly designate the elements that can be accessed by the users using the **SET ALLOWED METHODS** command. If *formula* calls methods that were not first “authorized” in the Formula editor using the **SET ALLOWED METHODS** command, a syntax error is generated and you will not be able to validate the dialog box.
- The formula editor is not associated with any menu bar by default. You need to install a standard **Edit** menu in the calling process if you want users to be able to benefit from cut/copy/paste shortcuts in the formula editor.

Keep in mind that when the dialog box is validated, the command does not execute the *formula*; it only validates and updates the contents of the variable. If you want to execute the *formula*, you must use the **EXECUTE FORMULA** command.

## Example

Displaying the Formula editor with the [Employees] table and without a pre-entered formula:

```
$myFormula:=""
EDIT FORMULA([Employees];$myFormula)
If(OK=1)
   APPLY TO SELECTION([Employees];EXECUTE FORMULA($myFormula))
End if
```

## System variables and sets

If the user validates the dialog box, the system variable OK is set to 1. If the user cancels the dialog box, the system variable OK is set to 0.

# EXECUTE FORMULA

EXECUTE FORMULA ( statement )

| Parameter | Type | | Description |
|---|---|---|---|
| statement | String | ➡ | Code to be executed |

## Description

**EXECUTE FORMULA** executes *statement* as a line of code. The statement string must be one line. If *statement* is an empty string, **EXECUTE FORMULA** does nothing.

The rule of thumb is that if the *statement* can be executed as a one-line method, then it will execute properly. Use **EXECUTE FORMULA** sparingly, as it slows down execution speed. In a compiled database, the line of code is not compiled. This means that *statement* will be executed, but it will not have been checked by the compiler at compilation time.

**Note:** Executing formulas in compiled mode can be optimized using a cache (see **Cache for formulas in compiled mode** below).

The statement can include the following elements:

- a Call to a project method
- a Call to a 4D command
- an Assignment

The formula can include process variables and interprocess variables. However, the statement cannot contain control of flow statements (If, While, etc.), because it must be in one line of code.

To ensure that the *statement* will be evaluated correctly regardless of the 4D language or version used, we recommend using the *token* syntax for elements whose name might vary between different versions (commands, tables, fields, constants). For example, to insert the **Current time** command, enter '**Current time:C178**'. For more information about this, refer to **Using tokens in formulas**.

### Cache for formulas in compiled mode

For optimization purposes, each formula executed by **EXECUTE FORMULA** in compiled mode can be stored in a dedicated cache in memory. The formula is cached in tokenized form. Once it is placed in the cache, its subsequent executions are highly optimized since the tokenization step is bypassed.

The cache size is zero by default (no cache); it needs to be created or adjusted using the **SET DATABASE PARAMETER** command. For example:

```
SET DATABASE PARAMETER(Number of formulas in cache;0) //no cache for formulas
SET DATABASE PARAMETER(Number of formulas in cache;3) //up to three formulas can be cached for each process
```

The **EXECUTE FORMULA** command uses the cache only when called from a compiled database or component.

## Example

You want to execute formulas including calls to 4D commands and tables. Since these elements could potentially be renamed, you can ensure correct execution in future versions of your application by using the token syntax as shown here:

```
EXECUTE FORMULA("Year of:C25 ([Products:5]Creation_Date:2])+$add")
```

# GET ALLOWED METHODS

GET ALLOWED METHODS ( methodsArray )

| Parameter | Type | | Description |
|---|---|---|---|
| methodsArray | String array | ⇦ | Array of method names |

## Description

The **GET ALLOWED METHODS** command returns, in *methodsArray*, the names of methods that can be used to write formulas. These methods are listed at the end of the list of commands in the editor.

By default, methods cannot be used in the Formula editor. Methods must be explicitly authorized using the **SET ALLOWED METHODS** command. If this command has not been executed, **GET ALLOWED METHODS** returns an empty array.

**GET ALLOWED METHODS** returns exactly what was passed to the **SET ALLOWED METHODS** command, i.e. a string array (the command creates and sizes the array). Also, if the wildcard (@) character is used to set a group of methods, the string containing the @ character is returned (and not the names of the methods of the group).

This command is useful for storing the settings of the current set of authorized methods before the execution of a formula in a specific context (for instance, a quick report).

## Example

This example authorizes a set of specific methods to create a report:

```
 `Store current parameters
GET ALLOWED METHODS(methodsArray)

 `Define methods for quick report
methodsarr_Reports{1}:="Reports_@"
SET ALLOWED METHODS(methodsarr_Reports)
QR REPORT([People];"MyReport")

 `Re-establish current parameters
SET ALLOWED METHODS(methodsArray)
```

# SET ALLOWED METHODS

SET ALLOWED METHODS ( methodsArray )

| Parameter | Type | | Description |
|---|---|---|---|
| methodsArray | String array | ➡ | Array of method names |

## Description

The **SET ALLOWED METHODS** command sets the methods that are displayed in the Formula editor for the current session. The designated methods will appear at the end of the list of commands and can be used in formulas. By default (if this command is not used), no methods are visible in the Formula editor. If a formula uses an unauthorized method name, a syntax error is generated and the formula cannot be validated.

Pass the name of an array containing the list of methods to offer in the Formula editor in the *methodsArray* parameter. The array must have been set previously.
You can use the wildcard character (@) in method names to define one or more authorized method groups.

If you would like the user to be able to call 4D commands that are unauthorized by default or plug-in commands, you must use specific methods that handle these commands.

**Note:** The mechanism for restricting access to commands and methods in the Formula editor can be disabled for all users or for the Designer and Administrator by means of an option on the "Security" page of the Database Settings. If the "Disabled for all" option is checked, the **SET ALLOWED METHODS** command will have no effect.

## Example

This example authorizes all methods starting with “formula” and the “Total_general” method in the Formula editor:

```
ARRAY STRING(15;methodsArray;2)
methodsArray{1}:="formula@"
methodsArray{2}:="Total_general"
SET ALLOWED METHODS(methodsArray)
```

# Graphs

GRAPH
GRAPH SETTINGS
*_o_GRAPH TABLE*

# GRAPH

GRAPH ( graphPicture ; graphNumber | graphSettings ; xLabels {; yElements} {; yElements2 ; ... ; yElementsN} )

| Parameter | Type | | Description |
|---|---|---|---|
| graphPicture | Picture variable | ⇒ | Picture variable |
| graphNumber \| graphSettings | Longint, Object | ⇒ | Longint: Graph type number, Object (64-bits only): Graph settings |
| xLabels | Array | ⇒ | Labels for the x-axis |
| yElements | Array | ⇒ | Data to graph (up to eight allowed) |

## Description

***Compatibility note**: Beginning with 4D v14, the GRAPH command only works with a picture variable as its first parameter. The obsolete syntax using a graph area (4D Chart) is no longer supported.*

**GRAPH** draws a graph for a picture variable on the basis of values coming from arrays.

The graphs generated by this command are drawn using the integrated SVG rendering engine. They have interface functions associated with picture variables: a context menu in Application mode (to let you choose, more particularly, the display format), scrollbars, and so on.

**Note:** SVG (Scalable Vector Graphics) is a graphics file format (.svg extension). Based on XML, this format is widespread and can be displayed more particularly in Web browsers. For more information, please refer to the following address: http://www.w3.org.Graphics/SVG/. The **SVG EXPORT TO PICTURE** command can also be used to take advantage of the integrated SVG engine.

In the *graphPicture* parameter, pass the name of the picture variable that displays the graph in the form.

The second parameter specifies the type of graph that will be drawn. You have two options:

- pass a *graphNumber* parameter of the *Longint* type (all versions of 4D): In this case, you must pass a number from 1 to 8. The graph types are described in Example 1. After a graph has been drawn, you can change the type by changing *graphNumber* and executing the **GRAPH** command again. You can then modify certain graph characteristics by calling the **GRAPH SETTINGS** command. See Example 1.
- pass a *graphSettings* parameter of the *Object* type (64-bit versions of 4D only, except for 64-bit versions of 4D Server on Windows): In this case, you must pass an object that contains the various graph properties to define. To do this, you can use the constants found in the "**Graph Parameters**" theme (see below). This syntax allows you to define the graph type along with its specific settings (legend, xmin, etc.) in a single call. This allows users to save the generated graphs as regular SVG pictures and makes it possible to display them using a standard browser like FireFox, Chrome, IE, or Safari (the graphs generated have better compliance with the SVG standard implemented in browsers). Furthermore, this syntax gives you access to various extra settings, which allow you to customize, for example, spacing between bars, margins, bar colors, etc. See Examples 2, 3 and 4. **Warning**: If you use this syntax, the **GRAPH SETTINGS** command must NOT be called.

The *xLabels* parameter defines the labels that will be used to label the x-axis (the bottom of the graph). This data can be of string, date, time, or numeric type. There should be the same number of array elements in *xLabels* as there are subrecords or array elements in each of the *yElements*.

The data specified by *yElements* is the data to graph. The data must be numeric. Up to eight data sets can be graphed. Pie charts graph only the first *yElements*.

### Automatic IDs

Specific IDs are assigned automatically to elements found in the SVG graph:

| IDs | Description |
| --- | --- |
| ID_graph_1 to ID_graph_8 | Columns, lines, areas... |
| ID_graph_shadow_1 to ID_graph_shadow_8 | Shadows for columns, lines, areas... |
| ID_bullet_1 to ID_bullet_8 | Points (*Line and Scatter graphs only*) |
| ID_pie_label_1 to ID_pie_label_8 | Pie chart labels (*Pie chart graphs only*) |
| ID_legend | Legend |
| ID_legend_1 to ID_legend_8 | Legend titles |
| ID_legend_border | Legend borders |
| ID_legend_border_shadow | Shadows for legend borders |
| ID_x_values | Values of X axis |
| ID_y_values | Values of Y axis |
| ID_y0_axis | Values of Z axis |
| ID_background | Background |
| ID_background_shadow | Background shadow |
| ID_x_grid | Grid on X axis |
| ID_x_grid_shadow | Shadow for grid on X axis |
| ID_y_grid | Grid on Y axis |
| ID_y_grid_shadow | Shadow for grid on Y axis |

### graphSettings attributes

When you use the *graphSettings* parameter, you pass an object that contains the various graph properties to define. You can use the following constants, found in the "**Graph Parameters**" constant theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Graph background color | String | graphBackgroundColor | **Possible values:** SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414" |
| Graph background opacity | String | graphBackgroundOpacity | **Possible values**: Integers, range 0-100<br>**Default value**: 100 |
| Graph background shadow color | String | graphBackgroundShadowColor | **Possible values**: SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414" |
| Graph bottom margin | String | bottomMargin | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph colors | String | colors | **Possible values**: Text array. Colors for each graph series.<br>**Default values**: Blue-green (#19BAC9), Yellow (#FFC338), Purple (#573E82), Green (#4FA839), Orange (#D95700), Blue (#1D9DF2), Yellow-green (#B5CF32), Red (#D43A26) |
| Graph column gap | String | columnGap | **Possible value**s: Longints<br>**Default value**: 12<br>Sets spacing between bars |
| Graph column width max | String | columnWidthMax | **Possible values**: Real numbers<br>**Default value**: 200 |
| Graph column width min | String | columnWidthMin | **Possible values**: Real numbers<br>**Default value**: 10 |
| Graph default height | String | defaultHeight | **Possible values**: Real numbers<br>**Default value**: 400. If graphType=7 (Pie), then default value = 600 |
| Graph default width | String | defaultWidth | **Possible values**: Real numbers<br>**Default value**: 600. If graphType=7 (Pie), then default value = 800 |
| Graph display legend | String | displayLegend | **Possible value**s: Boolean<br>**Default value**: True |
| Graph document background color | String | documentBackgroundColor | **Possible values**: SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414". When a graph saved as an SVG picture is opened elsewhere, the document background color is only displayed if the SVG rendering engine supports the *SVG tiny 1.2* norm (supported on IE, Firefox, but not on Chrome). |
| Graph document background opacity | String | documentBackgroundOpacity | **Possible values**: Integer, range 0-100 (default value: 100). When a graph saved as an SVG picture is opened elsewhere, the document background opacity is only displayed if the SVG rendering engine supports the *SVG tiny 1.2* norm (supported on IE, Firefox, but not on Chrome). |
| Graph font color | String | fontColor | **Possible values:** SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414" |
| Graph font size | String | fontSize | **Possible values**: Longints<br>**Default value**: 12. If graphType=7 (Pie), see Graph pie font size |
| Graph left margin | String | leftMargin | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph legend font color | String | legendFontColor | **Possible values**: SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414" |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Graph legend icon gap | String | legendIconGap | **Possible values**: Real numbers<br>**Default value**: Graph legend icon height/2 |
| Graph legend icon height | String | legendIconHeight | **Possible values**: Real numbers<br>**Default value**: 20 |
| Graph legend icon width | String | legendIconWidth | **Possible values**: Real numbers<br>**Default value**: 20 |
| Graph legend labels | String | legendLabels | **Possible values**: Text array. If missing, 4D displays icons without text. |
| Graph line width | String | lineWidth | **Possible values**: Real numbers<br>**Default value**: 2 |
| Graph pie font size | String | pieFontSize | **Possible values**: Real numbers<br>**Default value**: 16 |
| Graph pie shift | String | pieShift | **Possible values**: Real numbers<br>**Default value**: 8 |
| Graph plot height | String | plotHeight | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph plot radius | String | plotRadius | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph plot width | String | plotWidth | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph right margin | String | rightMargin | **Possible values**: Real numbers<br>**Default value**: 2 |
| Graph top margin | String | topMargin | **Possible values**: Real numbers<br>**Default value**: 2 |
| Graph type | String | graphType | **Possible values**: Longints [1 to 8] where 1 = bars, 2 = proportional, 3 = stacked, 4 = lines, 5 = surfaces, 6 = scatter, 7 = pie, 8 = pictures.<br>**Default value**: 1<br>If null, the graph is not drawn and no error message is displayed. If out of range, the graph is not drawn and an error message is displayed.<br>If you want to modify picture type graphs (value=8), you must copy the 4D/Resources/GraphTemplates/Graph_8_Pictures/ folder into the Resources folder of your database and perform the necessary modifications. Local picture files will be used instead of 4D files. There is no pattern for picture names; 4D sorts the files contained in the folder and assigns the first file to the first graph. These files can be of the SVG or image type. |
| Graph xGrid | String | xGrid | **Possible values**: Boolean<br>**Default value**: True.<br>Used only with proportional types 4 and 6. |
| Graph xMax | String | xMax | **Possible values**: Number, Date, Time (same type as *xLabels* parameter).<br>Only values lower than xMax are displayed on the graph. xMax is used only for graph types 4, 5, or 6 if xProp=true and if *xLabels* type is a number, date, or time. If missing or if xMin>xMax, 4D automatically calculates the xMax value. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Graph xMin | String | xMin | **Possible values**: Number, Date, Time (same type as *xLabels* parameter).<br>Only values higher than xMin are displayed on the graph. xMin is used only for graph types 4, 5, or 6 if xProp=true and if *xLabels* type is a number, date, or time. If missing or if xMin>xMax, 4D automatically calculates the xMin value. |
| Graph xProp | String | xProp | **Possible values**: Boolean<br>**Default value**: False<br>True for proportional x-axis; False for normal x-axis. xProp is used only for graph types 4, 5, or 6 |
| Graph yGrid | String | yGrid | **Possible values**: Boolean<br>**Default value**: True |
| Graph yMax | String | yMax | **Possible values**: Numbers<br>If missing, 4D automatically calculates the yMax value. |
| Graph yMin | String | yMin | **Possible values**: Numbers<br>If missing, 4D automatically calculates the yMin value. |

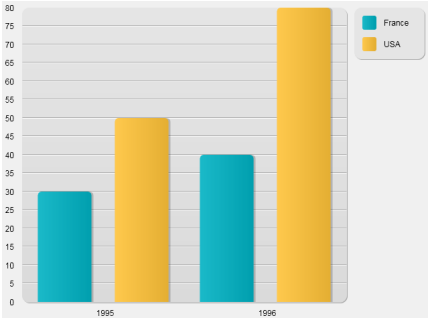
## Example 1

Syntax using *graphNumber*: The following example shows the different types of graphs that you can obtain. The code would be inserted in a form method or object method. It is not intended to be realistic, since the data is constant:

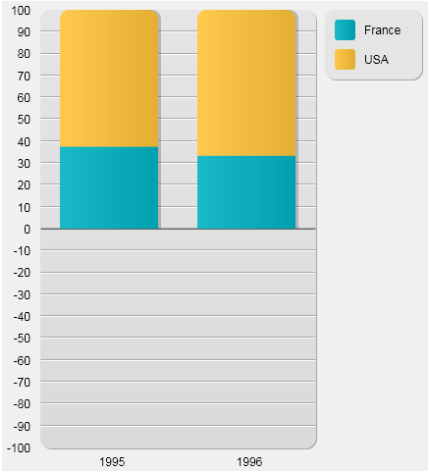
```
C_PICTURE(vGraph) //Variable of graph
ARRAY TEXT(X;2) //Create an array for the x-axis
X{1}:="1995" //X Label #1
X{2}:="1996" //X Label #2
ARRAY REAL(A;2) //Create an array for the y-axis
A{1}:=30 //Insert some data
A{2}:=40
ARRAY REAL(B;2) //Create an array for the y-axis
B{1}:=50 //Insert some data
B{2}:=80
vType:=1 //Initialize graph type
GRAPH(vGraph;vType;X;A;B) //Draw the graph
GRAPH SETTINGS(vGraph;0;0;0;0;False;False;True;"France";"USA") //Set the legends for the graph
```

The following figures show the resulting graph.

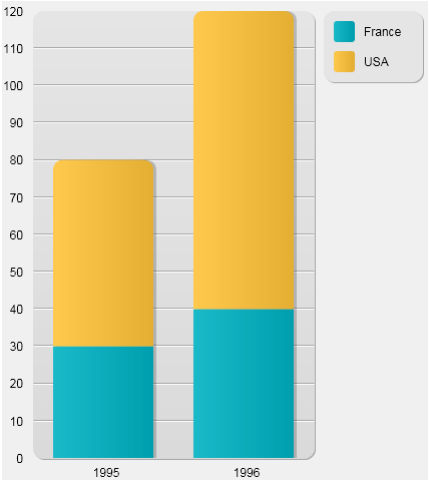
- With *vType* equal to 1, you obtain a **Column** graph:



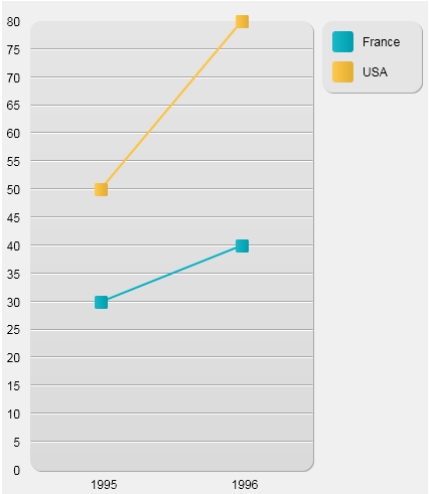
- With *vType* equal to 2, you obtain a **Proportional Column** graph:



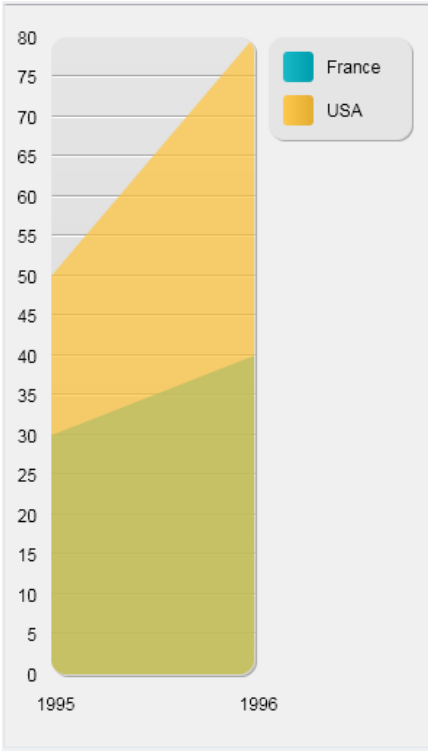

- With *vType* equal to 3, you obtain a **Stacked Column** graph:



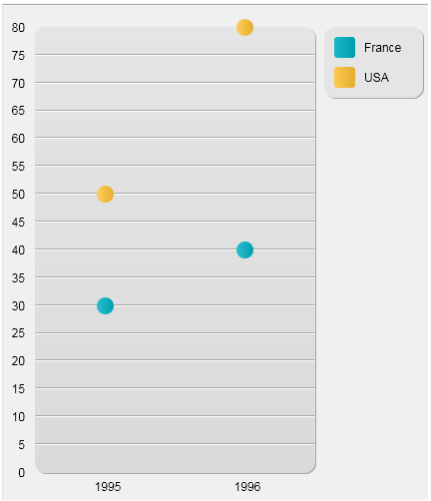

- With *vType* equal to 4, you obtain a **Line** graph:



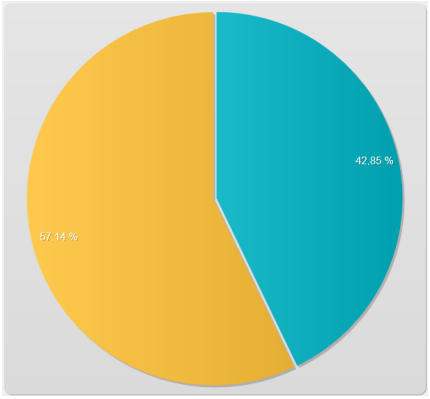
- With *vType* equal to 5, you obtain a **Area** graph:



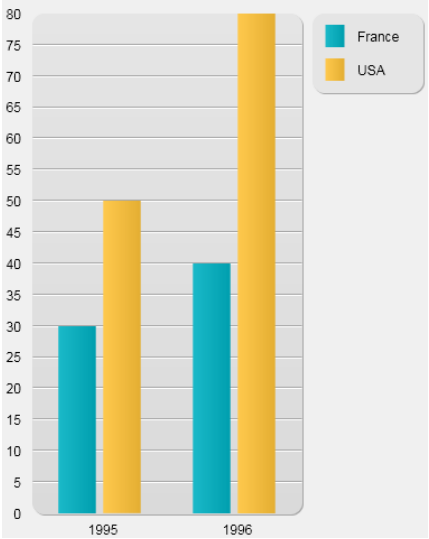

- With *vType* equal to 6, you obtain a **Scatter** graph:




- With *vType* equal to 7, you obtain a **Pie** graph:



- With *vType* equal to 8, you obtain a **Picture** graph:

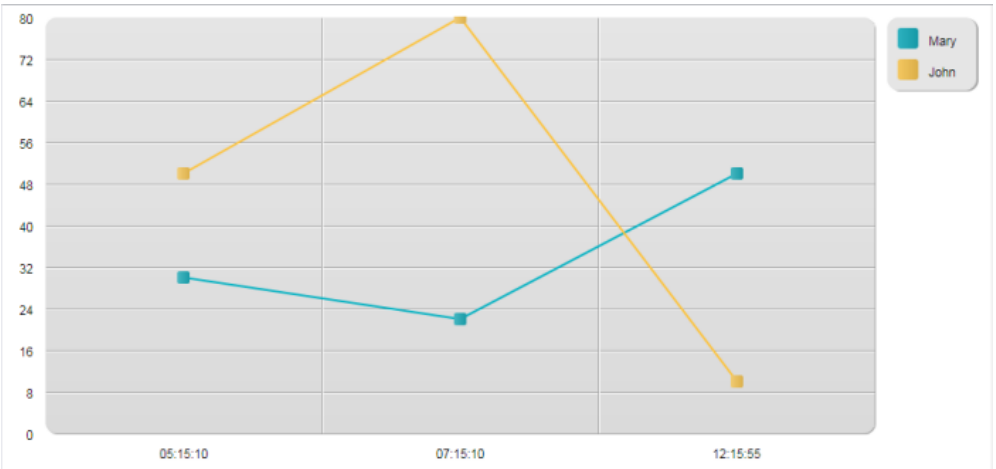



**Note:** Pictures are simple rectangles by default.

## Example 2

Syntax using *graphSettings*: In the following example, you draw a simple line graph based on time values:

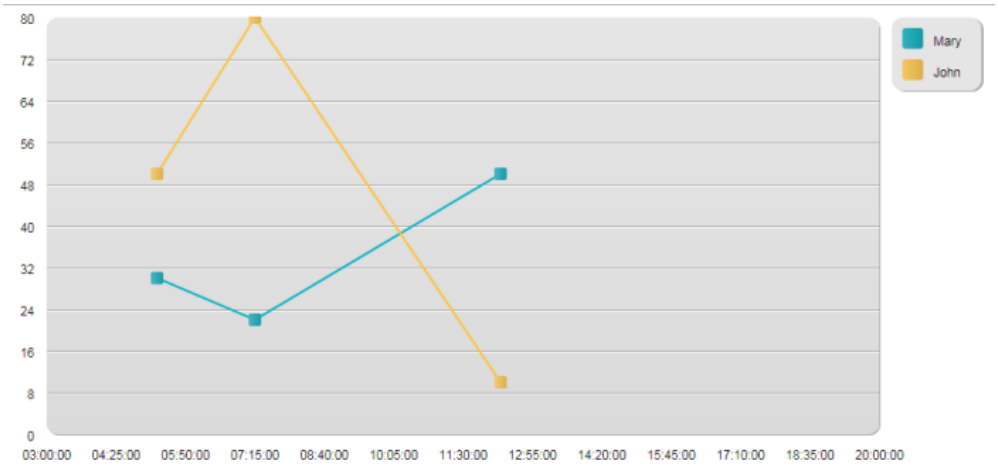
```
C_PICTURE(vGraph) //Graph variable
ARRAY TIME(X;3) //Create array for x-axis
X{1}:=?05:15:10? //X Label #1
X{2}:=?07:15:10? //X Label #2
X{3}:=?12:15:55? //X Label #3

ARRAY REAL(A;3) //Create array for y-axis
A{1}:=30 //Insert some data
A{2}:=22
A{3}:=50

ARRAY REAL(B;3) //Create another array for y-axis
B{1}:=50 //Insert some data
B{2}:=80
B{3}:=10

C_OBJECT(vSettings) //Initialize graph settings

OB SET(vSettings;Graph type;4) //Line type

ARRAY TEXT(aLabels;2) //Set legends for graph
aLabels{1}:="Mary"
aLabels{2}:="John"
OB SET ARRAY(vSettings;Graph legend labels;aLabels)

GRAPH(vGraph;vSettings;X;A;B) //Draw graph
```



## Example 3

With the same values, you can add custom settings to obtain a different view:

```
C_PICTURE(vGraph) //Graph variable
ARRAY TIME(X;3) //Create an array for the x-axis
X{1}:=?05:15:10? //X Label #1
X{2}:=?07:15:10? //X Label #2
X{3}:=?12:15:55? //X Label #3

ARRAY REAL(A;3) //Create an array for the y-axis
A{1}:=30 //Insert some data
A{2}:=22
A{3}:=50

ARRAY REAL(B;3) //Create another array for the y-axis
B{1}:=50 //Insert some data
B{2}:=80
B{3}:=10

C_OBJECT(vSettings) //initializing graph settings

OB SET(vSettings;Graph type;4) //type Line

ARRAY TEXT(aLabels;2) //Set the legends for the graph
aLabels{1}:="Mary"
aLabels{2}:="John"
OB SET ARRAY(vSettings;Graph legend labels;aLabels)

 //options
OB SET(vSettings;Graph xProp;True) //set proportional
OB SET(vSettings;Graph xGrid;False) //remove the vertical grid
OB SET(vSettings;Graph xMin;?03:00:00?) //define boundaries
OB SET(vSettings;Graph xMax;?20:00:00?)

GRAPH(vGraph;vSettings;X;A;B) //Draw the graph
```

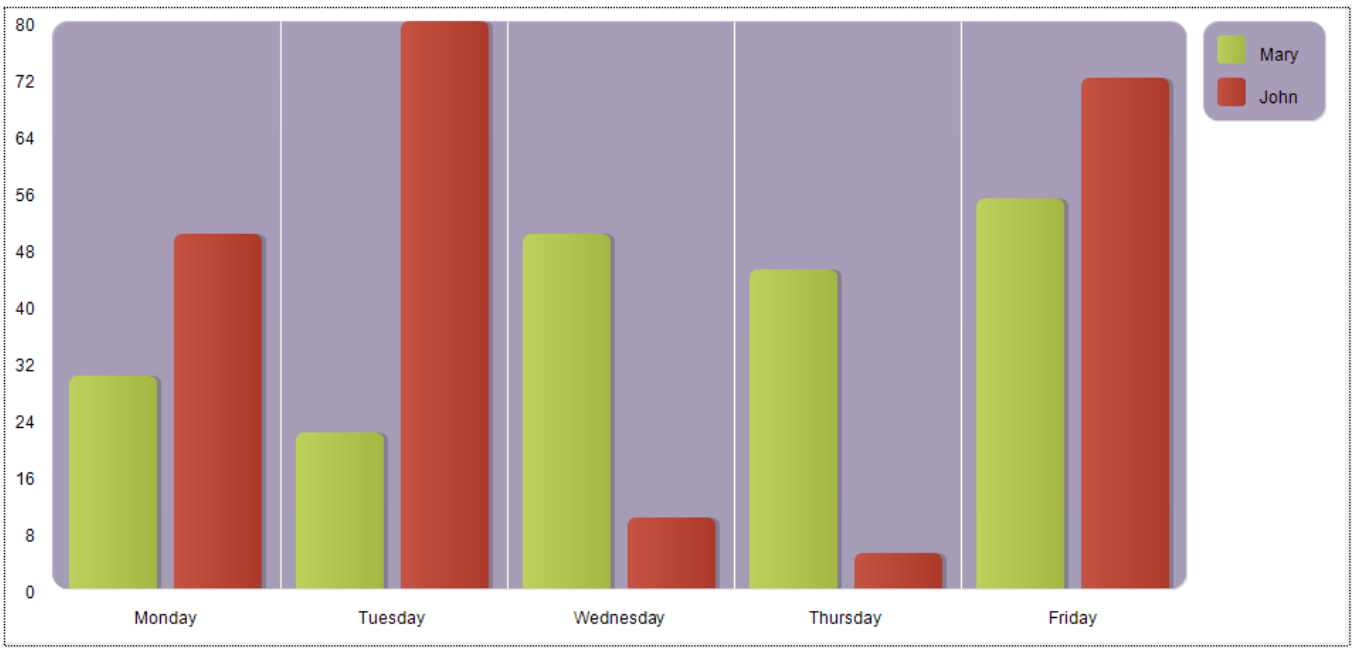



## Example 4

In this example, we customize a few settings:

```
C_PICTURE(vGraph) //Graph variable
ARRAY TEXT(X;5) //Create an array for the x-axis
X{1}:="Monday" //X Label #1
X{2}:="Tuesday" //X Label #2
X{3}:="Wednesday" //X Label #3
X{4}:="Thursday" //X Label #4
X{5}:="Friday" //X Label #5

ARRAY LONGINT(A;5) //Create an array for the y-axis
A{1}:=30 //Insert some data
```

```
A{2}:=22
A{3}:=50
A{4}:=45
A{5}:=55

ARRAY LONGINT(B;5) //Create another array for the y-axis
B{1}:=50 //Insert some data
B{2}:=80
B{3}:=10
B{4}:=5
B{5}:=72

C_OBJECT(vSettings) //initializing graph settings

OB SET(vSettings;Graph type;1) //type Bars

ARRAY TEXT(aLabels;2) //Set the legends for the graph
aLabels{1}:="Mary"
aLabels{2}:="John"
OB SET ARRAY(vSettings;Graph legend labels;aLabels)

 //options
OB SET(vSettings;Graph yGrid;False) //remove the vertical grid
OB SET(vSettings;Graph background color;"#573E82") //set a background color
OB SET(vSettings;Graph background opacity;40)
ARRAY TEXT($aTcols;2) //Set the colors for the graph
$aTcols{1}:="#B5CF32"
$aTcols{2}:="#D43A26"
OB SET ARRAY(vSettings;Graph colors;$aTcols)
GRAPH(vGraph;vSettings;X;A;B) //Draw the graph
```



# GRAPH SETTINGS

GRAPH SETTINGS ( graphPicture ; xmin ; xmax ; ymin ; ymax ; xprop ; xgrid ; ygrid ; title {; title2 ; ... ; titleN} )

| Parameter | Type | | Description |
|---|---|---|---|
| graphPicture | Picture variable | ➡ | Picture variable |
| xmin | Longint, Date, Time | ➡ | Minimum x-axis value for proportional graph (line or scatter plot only) |
| xmax | Longint, Date, Time | ➡ | Maximum x-axis value for proportional graph (line or scatter plot only) |
| ymin | Longint | ➡ | Minimum y-axis value |
| ymax | Longint | ➡ | Maximum y-axis value |
| xprop | Boolean | ➡ | TRUE for proportional x-axis; FALSE for normal x-axis (line or scatter plot only) |
| xgrid | Boolean | ➡ | TRUE for x-axis grid; FALSE for no x-axis grid (only if xprop is TRUE) |
| ygrid | Boolean | ➡ | TRUE for y-axis grid; FALSE for no y-axis grid |
| title | String | ➡ | Title(s) for graph legend(s) |

## Description

**GRAPH SETTINGS** changes the graph settings for graph displayed in a form. The graph must have already been defined using the **GRAPH** command. **GRAPH SETTINGS** has no effect on a pie chart. This command must be called in the same process as the form.

**Note:** You must not call this command if the graph was generated using the **GRAPH** command with the *Object* type *graphSettings* parameter. Refer to the description of the **GRAPH** command for more information.

The *xmin*, *xmax*, *ymin*, and *ymax* parameters all set the minimum and maximum values for their respective axes of the graph. If the value of any pair of these parameters is a null value (0, ?00:00:00?, or !00/00/00!, depending on the data type), the default graph values will be used. The *xmin* and *xmax* parameters are only taken into account for proportional graphs (*xprop* is **True**).

The *xprop* parameter turns on proportional plotting for line graphs (type 4) and scatter graphs (type 6). When TRUE, it will plot each point on the x-axis according to the point's value, and then only if the values are numeric, time, or date.

**Note:** With the 4D Server 64-bit version for OS X, the *xprop* parameter is also taken into account for area graphs (type 5).

The *xgrid* and *ygrid* parameters display or hide grid lines. A grid for the x-axis will be displayed only when the plot is a proportional scatter or line graph.

The *title* parameter(s) labels the legend.

## Example

See example for the **GRAPH** command.

# _o_GRAPH TABLE

_o_GRAPH TABLE
Does not require any parameters

## Description

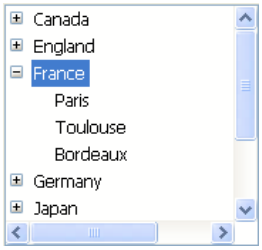
***Command deleted**: This command is no longer supported beginning with 4D v14*.

# Hierarchical Lists

- Managing Hierarchical Lists
- APPEND TO LIST
- CLEAR LIST
- Copy list
- Count list items
- DELETE FROM LIST
- Find in list
- GET LIST ITEM
- Get list item font
- GET LIST ITEM ICON
- GET LIST ITEM PARAMETER
- GET LIST ITEM PARAMETER ARRAYS
- GET LIST ITEM PROPERTIES
- GET LIST PROPERTIES
- INSERT IN LIST
- Is a list
- List item parent
- List item position
- LIST OF CHOICE LISTS
- Load list
- New list
- SAVE LIST
- SELECT LIST ITEMS BY POSITION
- SELECT LIST ITEMS BY REFERENCE
- Selected list items
- SET LIST ITEM
- SET LIST ITEM FONT
- SET LIST ITEM ICON
- SET LIST ITEM PARAMETER
- SET LIST ITEM PROPERTIES
- SET LIST PROPERTIES
- SORT LIST
- *_o_REDRAW LIST*

# Managing Hierarchical Lists

Hierarchical lists are form objects that can be used to display data as lists with one or more levels that can be expanded or collapsed.




In forms, hierarchical lists can be used for displaying or entering data. Each list item can contain up to 2 billion characters (maximum size of a text field) and be associated with an icon. They generally support clicks, double-clicks and keyboard navigation as well as drag and drop. It is possible to search the contents of a list (**Find in list** command).

## Creation and modification

Hierarchical lists can be created entirely by programming (via the **New list** or **Copy list** commands) or using lists defined in the List editor in Design mode (**Load list** command).

The contents and appearance of hierarchical lists are managed by programming using the commands of the "Hierarchical Lists" theme. Certain specific appearance characteristics can also be set using the generic commands of the **Object Properties** theme (see below).

You can associate hierarchical list references with form object choice lists (sources, required values and excluded values) dynamically using the **OBJECT SET LIST BY REFERENCE** or **OBJECT SET LIST BY NAME** commands. You can also associate choice lists defined in the List editor with form objects by means of the Property List.

## ListRef and object name

A hierarchical list is both a **language object** existing in memory and a **form object**.

The **language object** is referenced by an unique internal ID of the Longint type, designated by *ListRef* in this manual. This ID is returned by the commands that can be used to create lists: **New list**, **Copy list**, **Load list**, **BLOB to list**. There is only one instance of the language object in memory and any modification carried out on this object is immediately carried over to all the places where it is used.

The **form object** is not necessarily unique: there may be several representations of the same hierarchical list in the same form or in different ones. As with other form objects, you specify the object in the language using the syntax (*;"ListName", etc.).

You connect the hierarchical list "language object" with the hierarchical list "form object" by the intermediary of the variable containing the *ListRef* value. For example, if you write:

```
mylist:=New list
```

... you can simply associate the mylist variable name with the hierarchical list form object in the Property list so that it manages the language object whose *ListRef* is stored in mylist.

Each representation of the list has its own specific characteristics as well as sharing common characteristics with all the other representations.The following characteristics are specific to each representation of the list:

- The selection,
- The expanded/collapsed state of its items,
- The position of the scrolling cursor.

The other characteristics (font, font size, style, entry control, color, list contents, icons, etc.) are common to all the representations and cannot be modified separately.

Consequently, when you use commands based on the expanded/collapsed configuration or the current item, for example **Count list items** (when the final * parameter is not passed), it is important to be able to specify the representation to be used without any ambiguity.

You must use the *ListRef* ID with language commands when you want to specify the hierarchical list found in memory.

If you want to specify the representation of a hierarchical list object at the form level, you must use the object name (string type) in the command, via the syntax (*;"ListName", etc.).

This syntax is identical to that used in the commands of the "Object Properties" theme. It is accepted by most of the commands of the "Hierarchical Lists" theme that act on the properties of the lists (please see the description of the commands of this theme).

**Warning**: in the case of commands that set properties, the syntax based on the object name does not mean that only the form object specified will be modified by the command, but rather that the action of the command will be based on the state of this object. The common characteristics of hierarchical lists are always modified in all of their representations.

For example, if you pass the statement **SET LIST ITEM FONT**(*;"mylist1";*;thefont), you are indicating that you want to modify the font of the hierarchical list item associated with the mylist1 form object. The command will take the current item of the mylist1 object into account to specify the item to modify, but this modification will be carried over to all the representations of the list in all of the processes.

### Support of @

As with other object property management commands, it is possible to use the “@” character in the *ListName* parameter. As a rule, this syntax is used to designate a set of objects in the form. However, in the context of hierarchical list commands, this does not apply in every case. This syntax will have two different effects depending on the type of command:

- For commands that set properties, this syntax designates all the objects whose name corresponds (standard behavior). For example, the parameter "LH@" designates all objects of the hierarchical list type whose name begins with “LH.” These commands are:
  **DELETE FROM LIST**,
  **INSERT IN LIST**
  **SELECT LIST ITEMS BY POSITION**
  **SET LIST ITEM**
  **SET LIST ITEM FONT**
  **SET LIST ITEM ICON**
  **SET LIST ITEM PARAMETER**
  **SET LIST ITEM PROPERTIES**
- For commands retrieving properties, this syntax designates the first object whose name corresponds. These commands are:
  **Count list items**
  **Find in list**
  **GET LIST ITEM**
  **Get list item font**
  **GET LIST ITEM ICON**
  **GET LIST ITEM PARAMETER**
  **GET LIST ITEM PROPERTIES**
  **List item parent**
  **List item position**
  **Selected list items**

## Generic commands that can be used with hierarchical lists

It is possible to modify the appearance of a hierarchical list in a form using several generic 4D commands. You must pass to these commands either the object name of the hierarchical list (using the * parameter), or its variable name (standard syntax).

**Note:** In the case of hierarchical lists, the variable of the form contains the *ListRef* value. If you execute a command which modifies an attribute by passing the variable associated with the hierarchical list, it will not be possible to set the target list in the case of multiple representations. Therefore, only the object name allows you to differentiate individually between each different representation.

Here is a list of commands that can be used with hierarchical lists:
**OBJECT SET FONT**
**OBJECT SET FONT STYLE**
**OBJECT SET FONT SIZE**
**OBJECT SET COLOR**

**OBJECT SET FILTER**
**OBJECT SET ENTERABLE**
**OBJECT SET SCROLLBAR**
**OBJECT SET SCROLL POSITION**
**OBJECT SET RGB COLORS**

**Reminder:** Except for the **OBJECT SET SCROLL POSITION** command, these commands modify all the representations of the same list, even if you only specify a list via its object name

## Priority of property commands

Certain properties of hierarchical lists (for example, the Enterable attribute or the color) can be set in three different ways: via the Property list in Design mode, via a command of the "Object Properties" theme or via a command of the "Hierarchical Lists" theme.

When all three of these means are used to set list properties, the following order of priority is applied:

1. Commands of the "Hierarchical Lists" theme
2. Generic object property commands
3. Property list parameters

This principle is applied regardless of the order in which the commands are called. If an item property is modified individually via a hierarchical list command, the equivalent object property command will have no effect on this item even if it is called subsequently. For example, if you modify the color of an item via the **SET LIST ITEM PROPERTIES** command, the **OBJECT SET COLOR** command will have no effect on this item.

## Management of items by position or by reference

You can usually work in two ways with the contents of hierarchical lists: by position or by reference.

- When you work by position, 4D bases itself on the position in relation to the items of the list displayed on screen in order to identify them. The result will differ according to whether or not certain hierarchical items are expanded or collapsed. Note that in the case of multiple representations, each form object has its own configuration of expanded/collapsed items.
- When you work by reference, 4D bases itself on the *itemRef* ID number of the list items. Each item can thus be specified individually, regardless of its position or its display in the hierarchical list.

### Using item reference numbers (itemRef)

Each item of a hierarchical list has a reference number (*itemRef*) of the Longint type. This value is only intended for your own use: 4D simply maintains it.

**Warning:** You can use any type of Longint value as a reference number, except for 0. In fact, for most of the commands in this theme, the value 0 is used to specify the last item added to the list.

Here are a few tips for using reference numbers:

1. You do not need to identify each item with a unique number (beginner level).

- First example: you build a system of tabs by programming, for example, an address book. Since the system returns the number of the tab selected, you will probably not need more information than this. In this case, do not worry about item reference numbers: pass any value (except 0) in the *itemRef* parameter. Note that for an address book system, you can predefine a list A, B, ..., Z in Design mode, You can also create it by programming in order to eliminate any letters for which there are no records.

- Second example: while working with a database, you progressively build a list of keywords. You can save this list at the end of each session by using the **SAVE LIST** or **LIST TO BLOB** commands and reload it at the beginning of each new session using the **Load list** or **BLOB to list** commands. You can display this list in a floating palette; when each user clicks on a keyword in the list, the item chosen is inserted into the enterable area that is selected in the foreground process. You can also use drag and drop. In any case, the important thing is that you only process the item selected (by click or drag and drop), because the **Selected list items** (in the case of a click) and **DRAG AND DROP PROPERTIES** commands return the position of the item that you must process. When using this position value, you obtain the title of the item by means of the **GET LIST ITEM** command. Here again, you do not need to identify each item individually; you can pass any value (except 0) in the *itemRef* parameter.

2. You need to partially identify the list items (intermediary level).

You use the item reference number to store information needed when you must work with the item; this point is detailed in the example of the **APPEND TO LIST** command. In this example, we use the item reference numbers to store record numbers. However, we must be able to establish a distinction between items that correspond to the [Department] records and those that correspond to the [Employees] records.

3. You need to identify all the list items individually (advanced level).

You program an elaborate management of hierarchical lists in which you absolutely must be able to identify each item individually at every level of the list. A simple way of implementing this is to maintain a personal counter. Suppose that you create a hlList list using the **APPEND TO LIST** command. At this stage, you initialize a counter vhlCounter to 1. Each time you call **APPEND TO LIST** or **INSERT IN LIST**, you increment this counter (*vhlCounter:=vhlCounter+1*), and you pass the counter number as the item reference number. The trick consists in never decrementing the counter when you delete items — the counter can only increase. In this way, you guarantee the uniqueness of the item reference numbers. Since these numbers are of the Longint type, you can add or insert more than two billion items in a list that has been reinitialized... (however if you are working with such a great number of items, this usually means that you should use a table rather than a list.)

**Note**: If you use **Bitwise Operators**, you can also use item reference numbers for storing information that can be put into a Longint, i.e. 2 Integers, 4-byte values or, yet again, 32 Booleans.

### When do you need unique reference numbers?

In most cases, when using hierarchical lists for user interface purposes and when only dealing with the selected item (the one that was clicked or dragged), you will not need to use item reference numbers at all. Using **Selected list items** and **GET LIST ITEM** you have all you need to deal with the currently selected item. In addition, commands such as **INSERT IN LIST** and **DELETE FROM LIST** allow you to manipulate the list “relatively” with respect to the selected item.

Basically, you need to deal with item reference numbers when you want direct access to any item of the list programmatically and not necessarily the one currently selected in the list.

# APPEND TO LIST

APPEND TO LIST ( list ; itemText ; itemRef {; sublist ; expanded} )

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ⇒ | List reference number |
| itemText | String | ⇒ | Text of the new list item |
| itemRef | Longint | ⇒ | Unique reference number for the new list item |
| sublist | ListRef | ⇒ | Optional sublist to attach to the new list item |
| expanded | Boolean | ⇒ | Indicates if the optional sublist will be expanded or collapsed |

## Description

The **APPEND TO LIST** command appends a new item to the hierarchical list whose list reference number you pass in *list*.

You pass the text of the item in *itemText*. You can pass a string or text expression of up to 2 billion characters.

You pass the unique reference number of the item(of the Longint type) in *itemRef*. Although we qualify this item reference number as unique, you can actually pass the value you want. Refer to the **Managing Hierarchical Lists** section for more information about the *itemRef* parameter.

If you also want an item to have child items, pass a valid list reference to the child hierarchical list in *sublist*. In this case, you must also pass the *expanded* parameter. Pass **True** or **False** in this parameter so that the sublist is displayed expanded or collapsed respectively.

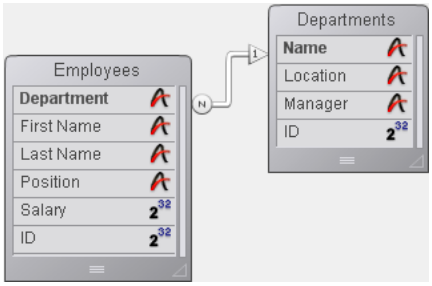
The list reference you pass in *sublist* must refer to an existing list. The existing list may be a one-level list or a list with sublists. If you do not want to attach a child list to the new item, omit the parameter or pass 0. Even though they are both optional, the *sublist* and *expanded* parameters must be passed jointly.

**Tips**

- To insert a new item in a list, use **INSERT IN LIST**. To change the text of an existing item or modify its child list as well as its expanded state, use **SET LIST ITEM**.
- To change the appearance of the new appended item use **SET LIST ITEM PROPERTIES**.

## Example

Here is a partial view of a database structure:




The [Departments] and [Employees] tables contain the following records:

| Name : | Location : | Manager : |
|---|---|---|
| Marine Biology | Aquarium B | Robert Masterson |
| Accounting | 3rd floor | Anne Weston |
| Sales | First floor-West | George Jackson |

| Department : | First Name : | Last Name : |
|---|---|---|
| Marine Biology | Andrew | Parker |
| Accounting | Jessica | Anders |
| Sales | Peter | Parker |
| Marine Biology | Jeffrey | Dalton |
| Accounting | Maria | Peterson |
| Sales | Jordan | Solomon |
| Sales | Patrick | McDonald |
| Marine Biology | Henry | Paulson |
| Sales | Marlo | Robertson |

You want to display a hierarchical list, named *hlList*, that shows the Departments, and for each Department, a child list that shows the Employees working in that Department. The object method of *hlList* is:

```
` hlList Hierarchical List Object Method

Case of

   :(Form event=On Load)
      C_LONGINT(hlList;$hSubList;$vlDepartment;$vlEmployee;$vlDepartmentID)
   ` Create a new empty hierarchical list
      hlList:=New list
   ` Select all the records from the [Departments] table
      ALL RECORDS([Departments])
   ` For each Department
      For($vlDepartment;1;Records in selection([Departments]))
   ` Select the Employees from this Department
         RELATE MANY([Departments]Name)
   ` How many are they?
         $vlNbEmployees:=Records in selection([Employees])
   ` Is there at least one Employee in this Department?
         If($vlNbEmployees>0)
   ` Create a child list for the Department item
            $hSubList:=New list
   ` For each Employee
            For($vlEmployee;1;Records in selection([Employees]))
   ` Add the Employee item to the sublist
   ` Note that ID field of the [Employees] record
   ` is passed as item reference number
               APPEND TO LIST($hSubList;[Employees]Last Name+", "+[Employees]First Name;[Employees]ID)
   ` Go the next [Employees] record
               NEXT RECORD([Employees])
            End for
         Else
   ` No Employees, no child list for the Department item
            $hSubList:=0
         End if
   ` Add the Department item to the main list
   ` Note that ID field of the [Departments] record
   ` is passed as item reference number. The bit #31
   ` of this item is forced to one so we'll be able
   ` to distinguish Department and Employee items. See note further
   ` below on why we can use this bit as supplementary information about
   ` the item.
         APPEND TO LIST(hlList;[Departments]Name;[Departments]ID?+31;$hSublist;$hSubList#0)
   ` Set the Department item in Bold to emphasize the hierarchy of the list
         SET LIST ITEM PROPERTIES(hlList;0;False;Bold;0)
   ` Go to the next Department
         NEXT RECORD([Departments])
      End for
   ` Sort the whole list in ascending order
      SORT LIST(hlList;>)
   ` Display the list using the Windows style
   ` and force the minimal line height to 14 Pts
      SET LIST PROPERTIES(hlList;Ala Windows;Windows node;14)

   :(Form event=On Unload)
   ` The list is no longer needed; do not forget to get rid of it!
      CLEAR LIST(hlList;*)

   :(Form event=On Double Clicked)
   ` A double-click occurred
   ` Get the position of the selected item
      $vlItemPos:=Selected list items(hlList)
   ` Just in case, check the position
      If($vlItemPos #0)
   ` Get the list item information
         GET LIST ITEM(hlList;$vlItemPos;$vlItemRef;$vsItemText;$vlItemSubList;$vbItemSubExpanded)
   ` Is the item a Department item?
         If($vlItemRef ?? 31)
   ` If so, it is a double-click on a Department Item
            ALERT("You double-clicked on the Department item "+Char(34)+$vsItemText+Char(34)+".")
         Else
```

```
      ` If not, it is a double-click on an Employee item
      ` Using the parent item ID to find the [Departments] record
                $vlDepartmentID:=List item parent(hlList;$vlItemRef)?-31
                QUERY([Departments];[Departments]ID=$vlDepartmentID)
      ` Tell where the Employee is working and to whom he or she is reporting
                ALERT("You double-clicked on the Employee item "+Char(34)+$vsItemText+Char(34)+
                " who is working in the Department "+Char(34)+[Departments]Name+Char(34)+
                " whose manager is "+Char(34)+[Departments]Manager+Char(34)+".")
             End if
          End if

   End case

      ` Note: 4D can store up to 1 billion records per table
      ` In our example, we use bit #31 of the unused high byte for
      ` distinguishing Employees and Departments items.
```

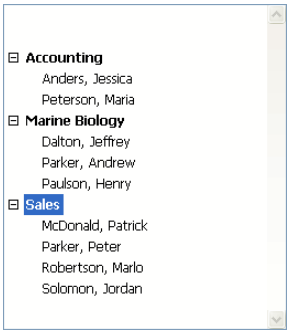
In this example, there is only one reason to distinguish [Departments] items and [Employees] items:

1. We store record IDs in the item reference numbers; therefore, we will probably end up with [Departments] items whose item reference numbers are the same as [Employees] items.

2. We use the **List item parent** command to retrieve the parent of the selected item. If we click on an [Employees] item whose associated record ID is #10, and if there is also a [Departments] item #10, the [Departments] item will be found first by **List item parent** when it browses the lists to locate the item with the item reference number we pass. The command will return the parent of the [Departments] item and not the parent of the [Employees] item.

Therefore, we made the item reference numbers unique, not because we wanted unique numbers, but because we needed to distinguish [Departments] and [Employees] records.

When the form is executed, the list will look like this:




**Note:** This example is useful for user interface purposes if you deal with a reasonably small number of records. Remember that lists are held in memory—do not build user interfaces with hierarchical lists containing millions of items.

# CLEAR LIST

CLEAR LIST ( list {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ➡ | List reference number |
| * | | ➡ | If specified, clear sublists from memory, if any. If omitted, sublists (if any) are not cleared. |

## Description

The **CLEAR LIST** command deletes the hierarchical list whose list reference number you pass in *list*.

Usually you will pass the optional * parameter, so all the sublists, if any, attached to items or subitems of the list will be deleted as well.

You do not need to clear a list attached to a form object via the Property List window. 4D loads and clears the list for you. On the other hand, each time you load, copy, extract from a BLOB, or create a list programmatically, call **CLEAR LIST** when you are through with the list.

To clear a sublist attached to an item (on any level) of another list currently displayed in a form, proceed as follows:

1. Call **GET LIST ITEM** on the parent item to get the list reference of the sublist.
2. Call **SET LIST ITEM** on the parent item to detach the sublist from the list item before clearing it.
3. Call **CLEAR LIST** to clear the sublist whose reference number you obtained with **GET LIST ITEM**.

## Example 1

Within a clean-up routine that clears all objects and data that you no longer need (i.e., when a window is closed and a form unloaded), you may end up clearing a hierarchical list that may have already been cleared, depending on the user actions within the form. Use **Is a list** to clear the list only if necessary:

```
  ` Extract of clean-up routine
If(Is a list(hlList))
   CLEAR LIST(hlList;*)
End if
```

## Example 2

See example for the **Load list** command.

## Example 3

See example for the **BLOB to list** command.

# Copy list

Copy list ( list ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ⇒ | Reference to list to be copied |
| Function result | ListRef | ⇐ | List reference number to duplicated list |

## Description

The **Copy list** command duplicates the list whose reference number you pass in *list,* and returns the list reference number of the new list.

After you have finished with the new list, call **CLEAR LIST** to delete it.

# Count list items

Count list items ( {* ;} list {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| * | Operator | ⇒ | If omitted (default): Return visible list items (expanded) If specified: Return all list items |
| Function result | Longint | ⇐ | Number of visible (expanded) list items (if 2nd * omitted) or Total number of list items (if 2nd * present) |

## Description

The **Count list items** command returns either the number of items currently "visible" or the total number of items in the list whose reference number or object name you pass in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with all the items (the second * is passed), you can use either syntax. Conversely, if you use several representations of the same list and work with the visible items (the second * is omitted), the syntax based on the object name is required since each representation can have its own expanded/collapsed configuration.

**Note:** If you use the @ character in the name of the list object and the form contains several lists that match with this name, the **Count list items** command will only apply to the first object whose name corresponds.

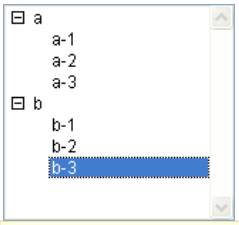
Use the second * parameter to determine which type of information will be returned. When this parameter is passed, the command returns the total number of items present in the list, regardless of whether it is expanded or collapsed.

When this parameter is omitted, the command returns the number of items that are visible, depending on the current expanded/collapsed state of the list and its sublists.

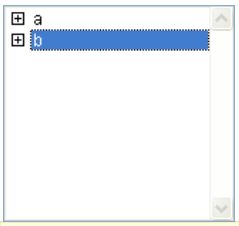
You apply this command to a list displayed in a form.

## Examples

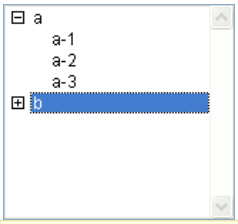
Here a list named *hList* shown in the Application environment:




```
$vlNbItems:=Count list items(hList) ` at this point $vlNbItems gets 8
$vlNbTItems:=Count list items(hList;*) `$vlNbTItems also gets 8
```




```
$vlNbItems:=Count list items(hList) ` at this point $vlNbItems gets 2
$vlNbTItems:=Count list items(hList;*) `$vlNbTItems still gets 8
```




```
$vlNbItems:=Count list items(hList) ` at this point $vlNbItems gets 5
$vlNbTItems:=Count list items(hList;*) `$vlNbTItems still gets 8
```

# DELETE FROM LIST

DELETE FROM LIST ( {* ;} list ; itemRef | * {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| itemRef \| * | Longint, Operator | ⇒ | Item reference number, or 0 for the last item added to the list or * for the currently selected list item |
| * | | ⇒ | If specified, erases sublists (if any) from memory If omitted, sublists (if any) are not erased |

## Description

The **DELETE FROM LIST** command deletes the item designated by the *itemRef* parameter of the list whose reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

If you pass * in *itemRef*, you delete the currently selected item in the list. You can also pass 0 in this parameter in order to request the deletion of the last item added to the list.

Otherwise, you specify the item reference number of the item you want to delete. If there is no item with the item reference number you passed, the command does nothing.

If you work with item reference numbers, build a list in which the items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, see the description of the **APPEND TO LIST** command.

No matter which item you delete, you should specify the optional * parameter to let 4D automatically delete the sublist attached to the item, if any. If you do not specify the * parameter, it is a good idea to have previously obtained the list reference number of the sublist (if any) attached to the item, so that you can delete it, if necessary, using the **CLEAR LIST** command.

## Example

The following code deletes the currently selected item of the list *hList*. If the item has an attached sublist, the sublist (as well as any sub-sublist) is deleted:

```
DELETE FROM LIST(hList;*;*)
```

# Find in list

Find in list ( {* ;} list ; value ; scope {; itemsArray {; *}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted) Name of list type object (if * passed) |
| value | String | ⇒ | Value to be searched for |
| scope | Integer | ⇒ | 0=Main list, 1=Sublist |
| itemsArray | Longint array | ⇐ | - If 2nd * omitted: array of positions of items found - If 2nd * passed: array of reference numbers of items found |
| * | Operator | ⇒ | - If omitted: use position of items - If passed: use reference number of items |
| Function result | Longint | ⇐ | - If 2nd * omitted: position of item found - If 2nd * passed: reference number of item found |

## Description

The **Find in list** command returns the position or reference of the first item of the *list* that is equivalent to the string passed in *value*. If several items are found, the function can also fill an *itemsArray* array with the position or reference of each item.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with item reference numbers (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with item positions (the second * is passed), the syntax based on the object name is required since the position of items can vary from one representation to another.

**Note:** If you use the @ character in the object name of the list and the form contains several lists that match this name, the **Find in list** command will be applied to the first object whose name corresponds.

The second * parameter can be used to indicate whether you want to work with the current positions of the items (in which case, this parameter is omitted) or with the absolute references of the items (in which case, it must be passed).

Pass the character strings to be searched for in *value*. The search will be of the “is exactly” type; in other words, searching for “wood” will not find “wooden.” However, you can use the wildcard character (@ ) to set up searches of the “begins with,” “ends with” or “contains” types.

The *scope* parameter is used to set whether the search must only be carried out at the first level of the *list* or whether it should include all the sublists. Pass 0 to limit the search to the first level of the list and 1 to extend it to all the sublists.

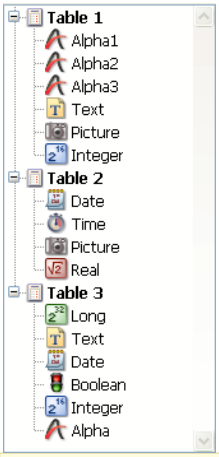
If you want to find out the position or number of all the items corresponding to *value*, pass a longint array in the optional *itemsArray* parameter. If necessary, the array will be created and resized by the command. The command will fill in the array with the positions (if the second * is omitted) or the reference numbers (if the second * is passed) of the items found.

Positions are expressed in relation to the top item of the main list, while taking into account the current expanded/collapsed state of the list and sublists.

If no item corresponds to the *value* searched for, the function returns 0 and the *itemsArray* array is returned empty.

## Example

Given the following hierarchical list:




```
$vlItemPos:=Find in list(hList;"P@";1;$arrPos)
  `$vlItemPos equals 6
  `$arrPos{1} equals 6 and $arrPos{2} equals 11
$vlItemRef:=Find in list(hList;"P@";1;$arrRefs;*)
  `$vlItemRef equals 7
  `$arrRefs{1} equals 7 and $arrRefs{2} equals 18
$vlItemPos:=Find in list(hList;"Date";1;$arrPos)
  `$vlItemPos equals 9
  `$arrPos{1} equals 9 and $arrPos{2} equals 16
$vlItemRefFind in list(hList;"Date";1;$arrRefs;*)
  `$vlItemRef equals 11
  `$arrRefs{1} equals 11 and $arrRefs{2} equals 23
$vlItemPos:=(hList;"Date";0;*)
  `$vlItemPos equals 0
```

# GET LIST ITEM

GET LIST ITEM ( {* ;} list ; itemPos | * ; itemRef ; itemText {; sublist ; expanded} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| itemPos \| * | Operator, Longint | ⇒ | Position of item in expanded/collapsed list(s) or * for the current item in the list |
| itemRef | Longint | ⇐ | Item reference number |
| itemText | String | ⇐ | Text of the list item |
| sublist | ListRef | ⇐ | Sublist list reference number (if any) |
| expanded | Boolean | ⇐ | If a sublist is attached: TRUE = sublist is currently expanded FALSE = sublist is currently collapsed |

## Description

The **GET LIST ITEM** command returns information about the item specified by *itemPos* of the list whose reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list, you can use either syntax. Conversely, if you use several representations of the same list, the syntax based on the object name is required since each representation can have its own expanded/collapsed configuration and its own current item.

**Note:** If you use the @ character in the name of the list object and the form contains several lists that match with this name, the **GET LIST ITEM** command will only apply to the first object whose name corresponds.

The position must be expressed relatively, using the current expanded/collapsed state of the list and its sublist. You pass a position value between 1 and the value returned by **Count list items**. If you pass a value outside this range, **GET LIST ITEM** returns empty values (0, "", etc.).

After the call, you retrieve:

- The item reference number of the item in *itemRef*.
- The text of the item in *itemText*.

If you passed the optional parameters *sublist* and *expanded*:

- *subList* returns the list reference number of the sublist attached to the item. If the item has no sublist, *subList* returns zero (0).
- If the item has a sublist, *expanded* returns TRUE if the sublist is currently expanded, and FALSE if it is collapsed.

## Example 1

*hList* is a list whose items have unique reference numbers. The following code programmatically toggles the expanded/collapsed state of the sublist, if any, attached to the current selected item:

```
$vlItemPos:=Selected list items(hList)
If($vlItemPos>0)
   GET LIST ITEM(hList;$vlItemPos;$vlItemRef;$vsItemText;$hSublist;$vbExpanded)
   If(Is a list($hSublist))
      SET LIST ITEM(hList;$vlItemRef;$vsItemText;$vlItemRef;$hSublist;Not($vbExpanded))
   End if
End if
```

## Example 2

Refer to the example of the **APPEND TO LIST** command.

# Get list item font

Get list item font ( {* ;} list ; itemRef | * ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted) or Name of list type object (if * passed) |
| itemRef \| * | Longint, Operator | ⇒ | Item reference number or 0 for the last item added to the list or * for the current item of the list |
| Function result | String | ⇐ | Font name |

## Description

The **Get list item font** command returns the current character font name of the item specified by the *itemRef* parameter of the list whose reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

**Note:** If you use the @ character in the object name of the list and the form contains several lists that match this name, the **Get list item font** command will be applied to the first object whose name corresponds.

You can pass a reference number in *itemRef*. If this number does not correspond to any item of the list, the command does nothing. You can also pass 0 in *itemRef* in order to get the font of the last item added to the list (using **APPEND TO LIST**).

Lastly, you can pass * in *itemRef*: in this case, the command will get the font of the current item of the list. If several items are selected manually, the current item is the one that was selected last. If no item is selected, the command does nothing.

# GET LIST ITEM ICON

GET LIST ITEM ICON ( {* ;} list ; itemRef | * ; icon )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted) or Name of list type object (if * passed) |
| itemRef \| * | Operator, Longint | ⇒ | Item reference number or 0 for the last item added to the list or * for the current item of the list |
| icon | Picture variable | ⇐ | Icon associated with item |

## Description

The **GET LIST ITEM ICON** command returns, in *icon*, the icon associated with the item whose reference number is passed in *itemRef* in the list whose reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

**Note:** If you use the @ character in the object name of the list and the form contains several lists that match this name, the **GET LIST ITEM ICON** command will be applied to the first object whose name corresponds.

You can pass a reference number in *itemRef*. If this number does not correspond to an item in the list, the command does nothing. You can also pass 0 in *itemRef* to indicate the last item added to the list (using **APPEND TO LIST**).
Lastly, you can pass * in *itemRef:* in this case, the command will apply to the current item of the list. If several items are selected manually, the current item is the one that was selected last. If no item is selected, the command does nothing.

Pass a picture variable in *icon*. After the command is executed, it will contain the icon associated with the item, regardless of the source of the icon (static picture, resource or picture expression).

If no icon is associated with the item, the icon variable is returned empty.

**Note:** When the icon associated with an item has been defined via a static reference (resource references or pictures from the picture library), it is possible to find out its number using the **GET LIST ITEM PROPERTIES** command.

# GET LIST ITEM PARAMETER

GET LIST ITEM PARAMETER ( {* ;} list ; itemRef | * ; selector ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted) or Name of list type object (if * passed) |
| itemRef \| * | Operator, Longint | ⇒ | Item reference number or 0 for the last item appended to the list or * for the current list item |
| selector | String | ⇒ | Parameter constant |
| value | String, Boolean, Real | ⇐ | Current value of parameter |

## Description

The **GET LIST ITEM PARAMETER** command is used to find out the current *value* of the *selector* parameter for the *itemRef* item of the hierarchical list whose reference or object name is passed in the *list* parameter.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second* is omitted), you can use either syntax. Conversely, if you use several representations of the same list and the second * is passed, the syntax based on the object name is required since each representation can have its own current item.

**Note:** If you use the @ character in the object name of the list and the form contains several lists that match this name, the **GET LIST ITEM PARAMETER** command will be applied to the first object whose name corresponds.

You can pass a reference number in *itemRef*. If this number does not correspond to an item in the list, the command does nothing. You can also pass 0 in *itemRef* to indicate the last item added to the list (using **APPEND TO LIST**).

Lastly, you can pass * in *itemRef*: in this case, the command is applied to the current item of the list. If several items are selected manually, the current item is the last one that was selected. If no item is selected, the command does nothing.

In *selector*, you can pass the Additional text constant (found in the "**Hierarchical Lists**" theme) or any custom value. For more information about the *selector* and *value* parameters, please refer to the description of the **SET LIST ITEM PARAMETER** command.

# GET LIST ITEM PARAMETER ARRAYS

GET LIST ITEM PARAMETER ARRAYS ( {* ;} list ; itemRef | * ; arrSelection {; arrValues} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, list is an object name (string)<br>If omitted, list is a list reference number |
| list | ListRef, String | ➡ | List reference number (if * omitted) or<br>Name of list type object (if * passed) |
| itemRef \| * | Longint, Operator | ➡ | Item reference number or<br>0 for the last item appended to the list or<br>* for the current list item |
| arrSelection | Text array | ⬅ | Array of parameter names |
| arrValues | Text array | ⬅ | Array of parameter values |

## Description

The **GET LIST ITEM PARAMETER ARRAYS** command lets you retrieve all the parameters in a single call (as well as, optionally, their values) that are associated with the *itemRef* item in the hierarchical list whose reference or object name is passed in the *list* parameter.

Parameters associated with items store additional information about each item. They are set using the **SET LIST ITEM PARAMETER** command.

If you pass the first optional * parameter, this indicates that *list* is an object name (string) corresponding to a list representation in the form. If you do not pass this parameter, this indicates that *list* is a hierarchical list reference (*ListRef*). If you use a single list representation or work with structural items (second * omitted), you can use either syntax. However, if you use several representations of the same list and work with the current item (second * passed), you must use the syntax based on the object name because each representation may have its own current item.

**GET LIST ITEM PARAMETER ARRAYS** returns parameters set for the *itemRef* item in the *arrSelectors* text array. When the *arrValues* text array is passed, the command uses it to return the values associated with these parameters.

*arrValues* must be a text type array. If you have associated values that are not Text (number or Boolean), they are converted to strings (True="1", False="0").

## Example

Given the following hierarchical list:

```
<>HL:=New list
$ID:=30
APPEND TO LIST(<>HL;"Martin";$ID)
 //5 parameters
SET LIST ITEM PARAMETER(<>HL;$ID;"Firstname";"Phil")
SET LIST ITEM PARAMETER(<>HL;$ID;"Birthday";"01/02/1978")
SET LIST ITEM PARAMETER(<>HL;$ID;"Male";True) //Boolean
SET LIST ITEM PARAMETER(<>HL;$ID;"Age";33) //number
SET LIST ITEM PARAMETER(<>HL;$ID;"City";"Dallas")
```

For more simplicity, the list was associated with a list object having the same name ("<>HL").
When the item "Martin" is selected in the list, you can retrieve its parameters by executing the following code:

```
ARRAY TEXT(arrParamNames;0)
GET LIST ITEM PARAMETER ARRAYS(*;"<>HL";*;arrParamNames)
 // arrParamNames{1} contains "Firstname"
 // arrParamNames{2} contains "Birthday"
 // arrParamNames{3} contains "Male"
 // arrParamNames{4} contains "Age"
 // arrParamNames{5} contains "City"
```

If you want to get the parameter values as well, you write:

```
ARRAY TEXT(arrParamNames;0)
ARRAY TEXT(arrParamValues;0)
GET LIST ITEM PARAMETER ARRAYS(*;"<>HL";*;arrParamNames;arrParamValues)
 // arrParamValues{1} contains "Phil"
 // arrParamValues{2} contains "01/02/1978"
 // arrParamValues{3} contains "1"
 // arrParamValues{4} contains "33"
 // arrParamValues{5} contains "Dallas"
```

# GET LIST ITEM PROPERTIES

GET LIST ITEM PROPERTIES ( {* ;} list ; itemRef | * ; enterable {; styles {; icon {; color}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| itemRef \| * | Operator, Longint | ⇒ | Item reference number, or 0 for last list item added, or * for the current list item |
| enterable | Boolean | ⇐ | TRUE = Enterable, FALSE = Non-enterable |
| styles | Longint | ⇐ | Font style for the item |
| icon | Longint | ⇐ | 'cicn' Mac OS-based resource ID, or 65536 + 'PICT' Mac OS-based resource ID, or 131072 + Picture Reference Number |
| color | Longint | ⇐ | RGB color value |

## Description

The **GET LIST ITEM PROPERTIES** command returns the properties of the item designated by the *itemRef* parameterwithin the list whose list reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

**Note:** If you use the @ character in the object name of the list and the form contains several lists matching this name, the **GET LIST ITEM PROPERTIES** command will be applied to the first object whose name corresponds.

In *itemRef*, you can pass either a reference number, or the value 0 in order to designate the last item added to the list, or * in order to designate the current item of the list. If several items are selected, the current item is the last one selected.

If you pass * and no item is selected or if there is no item with the item reference number that is passed, the command leaves the parameters unchanged.

If you work with item reference numbers, build a list in which items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, refer to the description of the command **APPEND TO LIST**.

After the call:

- *enterable* returns TRUE if the item is enterable.
- *styles* returns the font style of the item.
- *icon* returns the icon or picture assigned to the item, *0* if none.
- *color* returns the color of the text of the item specified.

**Note:** You can retrieve, in a picture variable, the icon associated with an item using the **GET LIST ITEM ICON** command.

For details about these properties, see the description of the command **SET LIST ITEM PROPERTIES**.

# GET LIST PROPERTIES

GET LIST PROPERTIES ( list ; appearance {; icon {; lineHeight {; doubleClick {; multiSelections {; editable}}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ⇒ | List reference number |
| appearance | Longint | ⇐ | Graphical style of the list 1 = Hierarchical list a la Macintosh 2 = Hierarchical list a la Windows |
| icon | Longint | ⇐ | 'cicn' Mac OS-based resource ID |
| lineHeight | Longint | ⇐ | Minimal line height expressed in pixels |
| doubleClick | Longint | ⇐ | Expand/Collapse sublist on double-click? 0 = Yes, 1= No |
| multiSelections | Longint | ⇐ | Multiple selections: 0 = No, 1 = Yes |
| editable | Longint | ⇐ | List editable by user: 0 = No, 1 = Yes |

## Description

The **GET LIST PROPERTIES** command returns information about the list whose reference number you pass in *list*.

The *appearance* parameter returns the graphical style of the list.

The *icon* parameter returns the resource IDs of the node icons displayed in the list.

The *lineHeight* parameter returns the minimal line height.

If *doubleClick* is set to 1, double-clicking on a parent list item does not provoke its child list to expand or to collapse. If *doubleClick* is set to 0, this behavior is active (defaut value).

If the *multiSelections* parameter is set to 0, multiple selections of items (manually or by programming) are not possible in the list. If it is set to 1, multiple selections are allowed.

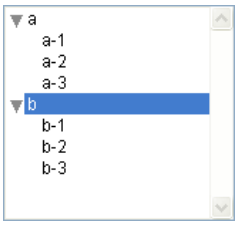
If the *editable* parameter is set to 1, the list is editable when it is displayed as a choice list in a record. If it is set to 0, the list is not editable.

These properties can be set using the **SET LIST PROPERTIES** command and/or in the Design environment List Editor, if the list was created there or saved using the **SAVE LIST** command.

For a complete description of the appearance, node icons, minimal line height and double-click management of a list, see the **SET LIST PROPERTIES** command.

## Example

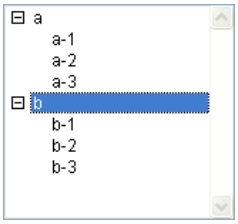
Given the list named *hList*, shown here in the Application environment:




The object method for a button:

```
   ` bMacOrWin button Object Method
GET LIST PROPERTIES(hList;$vlAppearance;$vlIcon;$vlLH;$vlClick;$vlSelect;$vlModif)
If($vlAppearance=Ala Macintosh)
   $vlAppearance:=Ala Windows
   $vlIcon:=Windows node
   $vlModif:=1
Else
   $vlAppearance:=A la Macintosh
   $vlIcon:=Macintosh node
   $vlModif:=1
End if
SET LIST PROPERTIES(hList;$vlAppearance;$vlIcon;$vlLH;$vlClick;$vlSelect;$vlModif)
```

This method lets you display the list as follows:


a
a-1
a-2
a-3
b
b-1
b-2
b-3

# INSERT IN LIST

INSERT IN LIST ( {* ;} list ; beforeItemRef | * ; itemText ; itemRef {; sublist ; expanded} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| beforeItemRef \| * | Longint, Operator | ⇒ | Item reference number or 0 for the last item added to the list or * for the currently selected list item |
| itemText | String | ⇒ | Text for the new list item |
| itemRef | Longint | ⇒ | Unique reference number for the new list item |
| sublist | ListRef | ⇒ | Optional sublist to attach to the new list item |
| expanded | Boolean | ⇒ | Indicates if the sublist will be expanded or collapsed |

## Description

The **INSERT IN LIST** command inserts the item designated by the *itemRef* parameter in the list whose reference number or object name you pass in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

The *beforeItemRef* parameter can be used to designate the item before which you wish to insert the new item:

- You can pass the value 0 in order to designate the last item added to the list. The newly inserted item will then become the selected item.
- You can pass * in order for the new item to be inserted before the currently selected item in the list. In this case, the newly inserted item will also become the selected item.
- Otherwise, if you want to insert an item before a specific item, you pass the item reference number of that item. In this case, the newly inserted item is not automatically selected. If there is no item with the corresponding item reference number, the command does nothing.

You pass the text and the item reference number of the new item in *itemText* and *itemRef*.

If you want for the item to include subitems, pass a valid list reference number in the *sublist* parameter. In this case, you must also pass the *expanded* parameter. Pass either **True** or **False** in this parameter so that this sublist is displayed either expanded or collapsed respectively.

## Example

The following code inserts an item (with no attached sublist) just before the currently selected item in the *hList* list:

```
vlUniqueRef:=vlUniqueRef+1
INSERT IN LIST(hList;*;"New Item";vlUniqueRef)
```

# Is a list

Is a list ( list ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ⇒ | ListRef value to be tested |
| Function result | Boolean | ⇐ | TRUE if list is a hierarchical list FALSE if list is not a hierarchical list |

## Description

The **Is a list** command returns TRUE if the value you pass in *list* is a valid reference to a hierarchical list. Otherwise, it returns FALSE.

## Example 1

See example for the **CLEAR LIST** command.

## Example 2

See examples for the **DRAG AND DROP PROPERTIES** command.

# List item parent

List item parent ( {* ;} list ; itemRef | * ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| itemRef \| * | Operator, Longint | ⇒ | Item reference number or 0 for the last item added to the list or * for the current item in the list |
| Function result | Longint | ⇐ | Item reference number of parent item or 0 if none |

## Description

The **List item parent** command returns the item reference number of a parent item.

Pass the reference number or object name of the list in *list* .

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

**Note:** If you use the @ character in the object name of the list and the form contains several lists matching this name, the **List item parent** command will be applied to the first object whose name corresponds.

You pass the item reference number of an item in the list or 0 or yet again *, in *itemRef*. If you pass 0, the command applies to the last item added to the list. If you pass *, the command applies to the current item of the list. If several items have been selected manually, the current item is the last one selected.

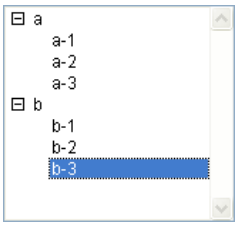
In return, if the corresponding item exists in the list and if this item is in a sublist (and therefore has a parent item), you obtain the item reference number of the parent item.

If there is no item with the item reference number you passed, or if you have passed * and no item is selected, or if the item has no parent, **List item parent** returns 0 (zero).

If you work with item reference numbers, be sure to build a list in which the items have unique reference numbers; otherwise you will not be able to distinguish the items. For more information, see the description of the **APPEND TO LIST** command.

## Example

Given the list named *hList* shown here in the Application environment:




The item reference numbers are set as follows:

| Item | Item Reference Number |
|---|---|
| *a* | *100* |
| *a - 1* | *101* |
| *a - 2* | *102* |
| *b* | *200* |
| *b - 1* | *201* |
| *b - 2* | *202* |
| *b - 3* | *203* |

- In the following code, if the item “b - 3” is selected, the variable *$vlParentItemRef* gets 200, the item reference number of the item “b”:

```
$vlItemPos:=Selected list items(hList)
GET LIST ITEM(hList;$vlItemPos;$vlItemRef;$vsItemText)
$vlParentItemRef:=List item parent(hList;$vlItemRef) ` $vlParentItemRef gets 200
```

- If the item “a - 1” is selected, the variable *$vlParentItemRef* gets 100, the item reference number of the item “a”.
- If the item “a” or “b” is selected, the variable *$vlParentItemRef* gets 0, because these items have no parent item.

# List item position

List item position ( {* ;} list ; itemRef ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| itemRef | Longint | ⇒ | Item reference number |
| Function result | Longint | ⇐ | Item position in expanded/collapsed lists |

## Description

The **List item position** command returns the position of the item whose item reference number is passed in *itemRef*, within the list whose list reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list, you can use either syntax. Conversely, if you use several representations of the same list, the syntax based on the object name is required since each representation can have its own expanded/collapsed configuration.

**Note:** If you use the @ character in the object name of the list and the form contains several lists matching this name, the **List item position** command will be applied to the first object whose name corresponds.

**Note:** Unlike the other commands of this theme, with this command it is not possible to pass the value 0 in *itemRef* to designate the last item added.

The position is expressed relative to the top item of the main list, using the current expanded/collapsed state of the list and its sublist.

The result is therefore a number between 1 and the value returned by **Count list items**.

If the item is not visible because it is located in a collapsed list, **List item position** expands the appropriate list to make the item visible.

If the item does not exist, **List item position** returns 0.

# LIST OF CHOICE LISTS

LIST OF CHOICE LISTS ( numsArray ; namesArray )

| Parameter | Type | | Description |
|---|---|---|---|
| numsArray | Longint array | ⇦ | Numbers of choice lists |
| namesArray | Text array | ⇦ | Names of choice lists |

## Description

The **LIST OF CHOICE LISTS** command returns, in the synchronized *numsArr* and *namesArr* arrays, the numbers and names of the choice lists defined by the list editor in Design mode.

The numbers of choice lists correspond to their order of creation. In the list editor, choice lists are displayed in alphabetical order.

# Load list

Load list ( listName ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| listName | String | ⇒ | Name of a list created in the Design environment List Editor |
| Function result | ListRef | ⇐ | List reference number of newly created list |

## Description

**Load list** creates a new hierarchical list whose contents are copied from the list and whose name you pass in *listName*. It then returns the list reference number to the newly created list.

To find out the lists specified in the database, use the **LIST OF CHOICE LISTS** command.

To make sure that the list specified by *listName* exists, use the **Is a list** function.

Note that the new list is a copy of the list defined in the Design environment. Consequently, any modifications made to the new list will not affect the list defined in the Design environment. Conversely, any subsequent modifications made to the list defined in the Design environment will not affect the list that you just created.

If you modify the newly created list and want to permanently save the changes, call **SAVE LIST**.

Remember to call **CLEAR LIST** in order to delete the newly created list when you have finished with it. Otherwise, it will stay in memory until the end of the working session or until the process in which it was created ends or is aborted.

**Tip:** If you associate a list with a form object (hierarchical list, tab control, or hierarchical pop-up menu) using the Choice List property in the Property List window, you do not need to call **Load list** or **CLEAR LIST** from the method of the object. 4D loads and clears the list automatically for you.

## Example

You create a database for the international market and you need to switch to different languages while using the database. In a form, you present a hierarchical list, named *hlList*, that proposes a list of standard options. In the Design environment, you have prepared various lists, such as “Std Options US” for the English version, “Std Options FR” for the French version, “Std Options SP” for the Spanish version, and so on. In addition, you maintain an interprocess variable, named *◊gsCurrentLanguage*, where you store a 2-character language code, such as “US” for the English version, “FR” for the French version, “SP” for the Spanish version, and so on. To make sure that your list will always be loaded using the current selected language, you can write:

```
  ` hlList Hierarchical List Object Method
Case of
   :(Form event=On Load)
      C_LONGINT(hlList)
      hlList:=Load list("Std Options"+◊gsCurrentLanguage)
   :(Form event=On Unload)
      CLEAR LIST(hlList;*)
End case
```

# New list

New list -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | ListRef | | List reference number |

## Description

**New list** creates a new, empty hierarchical list in memory and returns its unique list reference number.

**WARNING:** Hierarchical lists are held in memory. When you are finished with a hierarchical list, it is important to dispose of it and free the memory, using the command **CLEAR LIST**.

Several other commands allow you to create hierarchical lists:

- **Copy list** duplicates a list from an existing list.
- **Load list** creates a list by loading a Choice List created (manually or programmatically) in the Design enviornment List Editor.
- **BLOB to list** creates a list from the contents of a BLOB in which a list was previously saved.

After you have created a hierarchical list using **New list**, you can:

- Add items to that list, using the command **APPEND TO LIST** or **INSERT IN LIST**.
- Delete items from that list, using the command **DELETE FROM LIST**.

## Example

See example for the **APPEND TO LIST** command.

## SAVE LIST

SAVE LIST ( list ; listName )

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ⇒ | List reference number |
| listName | String | ⇒ | Name of the list as it will appear in the Design environment List Editor |

### Description

The **SAVE LIST** command saves the list whose reference number you pass in *list*, within the Design environment List Editor, under the name you pass in *listName*.

If there is already a list with this name, its contents are replaced.

# SELECT LIST ITEMS BY POSITION

SELECT LIST ITEMS BY POSITION ( {* ;} list ; itemPos {; positionsArray} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| itemPos | Longint | ⇒ | Position of item in expanded/collapsed list(s) |
| positionsArray | Longint array | ⇒ | Array of the positions in the expanded/collapsed list(s) |

## Description

The **SELECT LIST ITEMS BY POSITION** command selects the item(s) whose position is passed in *itemPos* and, optionally, in *positionsArray* within the list whose reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list, you can use either syntax. Conversely, if you use several representations of the same list, the syntax based on the object name is required since each representation can have its own expanded/collapsed configuration.

**Note:** If you use the @ character in the name of the list object and the form contains several lists that match with this name, the **SELECT LIST ITEMS BY POSITION** command will only apply to the first object whose name corresponds.

The position of items is always expressed using the current expanded/collapsed state of the list and its sublists. You pass a position value between 1 and the value returned by **Count list items**. If you pass a value outside this range, no item is selected.

If you do not pass the *positionsArray* parameter, the *itemPos* parameter represents the position of the item to be selected.

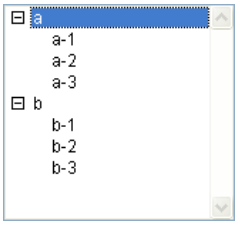
The optional *positionsArray* parameter lets you select several items simultaneously within the *list*. In *positionsArray*, you must pass an array where each line indicates the position of an item to be selected.

When you pass this parameter, the item designated by the *itemPos* parameter sets the new current item of the list among the resulting selection. It may or may not belong to the set of items defined by the array. The current item is, more particularly, the one that is edited if the **EDIT ITEM** command is used.

**Note:** In order for several items to be selected simultaneously in a hierarchical list (manually or by programming), the *multiSelections* property must have been enabled for this list. This property is set using the **SET LIST PROPERTIES** command.

## Example

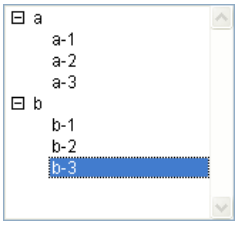
Given the hierarchical list named *hList,* shown here in the Application environment:




After the execution of this code:

```
SELECT LIST ITEMS BY POSITION(hList;Count list items(hList))
```

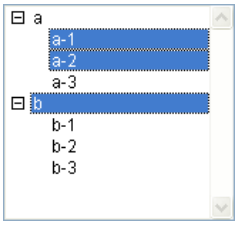
The last visible list item is selected:




After execution of the following lines of code:

```
SET LIST PROPERTIES(hList;0;0;18;0;1)
 `It is imperative to pass 1 as the last parameter in order to allow multiple selections
ARRAY LONGINT($arr;3)
$arr{1}:=2
$arr{2}:=3
$arr{3}:=5
SELECT LIST ITEMS BY POSITION(hList;3;$arr)
 `The 3rd item is designated as the current item
```

... the 2nd, 3rd and 5th items of the hierarchical list are selected:



# SELECT LIST ITEMS BY REFERENCE

SELECT LIST ITEMS BY REFERENCE ( list ; itemRef {; refArray} )

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ➡ | List reference number |
| itemRef | Longint | ➡ | Item reference number or 0 for the last item added to the list |
| refArray | Longint array | ➡ | Array of item reference numbers |

## Description

The **SELECT LIST ITEMS BY REFERENCE** command selects the item(s) whose item reference number is passed in *itemRef* and, optionally, in *refArray*, within the list whose reference number is passed in *list*.

If there is no item with the item reference number you passed, the command does nothing.

If an item is not currently visible (i.e., it is located in a collapsed sublist), the command expands the required sublist(s) so that it becomes visible.

If you do not pass the *refArray* parameter, the *itemRef* parameter represents the reference of the item to be selected. If the item number does not correspond to an item in the list, the command does nothing. You can also pass the value 0 in this parameter in order to designate the last item added to the list.

The optional *refArray* parameter lets you select several items simultaneously within the list. In *refArray*, you must pass an array where each line indicates the fixed reference of an item to be selected.

In this case, the item designated by the *itemRef* parameter sets the new current item of the list among the resulting selection. It may or may not belong to the set of items defined by the array. The current item is, more particularly, the one that is edited if the **EDIT ITEM** command is used.

**Note:** In order for several items to be selected simultaneously in a hierarchical list (manually or by programming), the *multiSelections* property must have been enabled for this list. This property is set using the **SET LIST PROPERTIES** command.

If you work with item reference numbers, be sure to build a list in which the items have unique reference numbers; otherwise you will not be able to distinguish them. For more information, see the description of the **APPEND TO LIST** command.

## Example

*hList* is a list whose items have unique reference numbers. The following object method for a button selects the parent item (if any) of the currently selected item:

```
$vlItemPos:=Selected list items(hList) ` Get position of selected item
GET LIST ITEM(hList;$vlItemPos;$vlItemRef;$vsItemText)
 ` Get item ref number of selected item
$vlParentItemRef:=List item parent(hList;$vlItemRef)
 ` Get item ref. number of parent item (if any)
If($vlParentItemRef>0)
   SELECT LIST ITEM BY REFERENCE(hList;List item parent(hList;$vlItemRef)) ` Select the parent item
End if
```

# Selected list items

Selected list items ( {* ;} list {; itemsArray {; *}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| itemsArray | Longint array | ⇐ | If 2nd * omitted: Array contains the positions of selected items in the list(s) If 2nd * passed: Array contains the selected item references |
| * | Operator | ⇒ | If omitted: Item position(s) If passed: Item reference(s) |
| Function result | Longint | ⇒ | If 2nd * omitted: Position of current selected list item in expanded/collapsed list(s) If 2nd * passed: Reference of the selected item |

## Description

The **Selected list items** command returns the position or reference of the selected item in the list whose reference number or object name you pass in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with item references (the second * is passed), you can use either syntax. Conversely, if you use several representations of the same list and work with the item positions (the second * is omitted), the syntax based on the object name is required since each representation can have its own expanded/collapsed item configuration.

**Note:** If you use the @ character in the name of the list object and the form contains several lists that match with this name, the **Selected list items** command will only apply to the first object whose name corresponds.

In the case of multiple selection, the command can also return in the *itemsArray* array, the position or reference of each item selected. You apply this command to a list displayed in a form to detect which item(s) the user has selected.

The second * parameter lets you indicate whether you want to work with current item positions (in this case, the * parameter should be omitted) or with fixed item references (in this case, the * parameter must be used).

You can pass a longint array in the *itemsArray* parameter. If necessary, the array will be created and resized by the command. Once the command has been executed, *itemsArray* will contain:

- the position of each item selected relative to the expanded/collapsed state of the list(s) if the * parameter is omitted.
- the fixed reference of each item selected if the * parameter is passed.
  If no items have been selected, the array is returned empty.

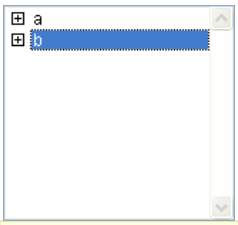
**Note:** In the event of multiple selections, the command returns the position or reference of the current item of *list*. The current item is the last item clicked by the user (manual selections) or the item set by the **SELECT LIST ITEMS BY POSITION** or **Selected list items** commands (programmed selection).

If the list has sublists, you apply the command to the main list (the one actually defined in the form), not one of its sublists. The positions are expressed relative to the top item of the main list, using the current expanded/collapsed state of the list and its sublist.

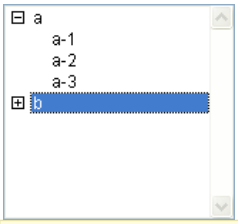
In any case, if no items are selected, the function returns 0.

## Example

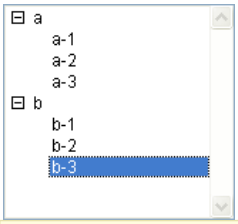
Here a list named *hList,* shown in the Application environment:



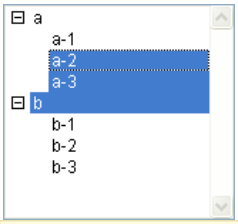

```
$vlItemPos:=Selected list items(hList) ` at this point $vlItemPos gets 2
```



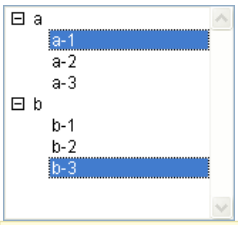

```
$vlItemPos:=Selected list items(hList) ` at this point $vlItemPos gets 5
$vlItemRef:=Selected list items(hList;*) ` $vlItemRef gets 200 (for instance)
```




```
$vlItemPos:=Selected list items(hList) ` at this point $vlItemPos gets 8
$vlItemRef:=Selected list items(hList;*) ` $vlItemRef gets 203 (for instance)
```




```
$vlItemPos:=Selected list items(hList;$arrPos) ` at this point, $vlItemPos gets 3
 ` $arrPos{1} gets 3, $arrPos{2} gets 4 and $arrPos{3} gets 5
```




```
$vlItemRef:=Selected list items(hList;$arrRefs;*) ` $vlItemRef gets 203 (for instance)
 ` $arrRefs{1} gets 101, $arrRefs{2} gets 203 (for instance)
```

# SET LIST ITEM

SET LIST ITEM ( {* ;} list ; itemRef | * ; newItemText ; newItemRef {; sublist ; expanded} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted), or Name of list type object (if * passed) |
| itemRef \| * | Operator, Longint | ⇒ | Item reference number, or 0 for last item appended to the list, or * for the current item in the list |
| newItemText | String | ⇒ | New item text |
| newItemRef | Longint | ⇒ | New item reference number |
| sublist | ListRef | ⇒ | New sublist attached to item, or 0 for no sublist (detaching current one, if any), or -1 for no change |
| expanded | Boolean | ⇒ | Indicates if the optional sublist will be expanded or collapsed |

## Description

The **SET LIST ITEM** command modifies the item designated by the *itemRef* parameter within the list whose reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

You can pass a reference number in *itemRef*. If there is no item with the item reference number you passed, the command does nothing. You can optionally pass *0* in *itemRef* to designate the last item added to the list using **APPEND TO LIST**.

Lastly, you can pass * in *itemRef*: in this case, the command will apply to the current item of the list. If several items are selected manually, the current item is the one that was selected last. If no item is selected, the command does nothing.

If you work with item reference numbers, build a list in which the items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, refer to the **Managing Hierarchical Lists** section.

You pass the new text for the item in *newItemText*. To change the item reference number, pass the new value in *newItemRef*; otherwise, pass the same value as *itemRef*.

To attach a list to the item, pass the list reference number in *subList*. In this case, you also specify if the newly sublist is expanded by passing TRUE in *expanded*; otherwise, pass FALSE.

To detach a sublist already attached to the item, pass *0* (zero) in *sublist*. In this case, it is a good idea to have previously obtained the reference number of that list using **APPEND TO LIST**, so you can later delete the sublist using **CLEAR LIST**, if you no longer need it.

If you do not want to change the sublist property of the item, pass *-1* in *sublist*.

**Note:** Even if they are optional, both the *sublist* and *expanded* parameters must be passed jointly.

## Example 1

*hList* is a list whose items have unique reference numbers. The following object method for a button adds a child item to the current selected list item.

```
$vlItemPos:=Selected list items(hList)
If($vlItemPos>0)
   GET LIST ITEM(hList;$vlItemPos;$vlItemRef;$vsItemText;$hSublist;$vbExpanded)
   $vbNewSubList:=Not(Is a list($hSublist))
   If($vbNewSubList)
      $hSublist:=New list
   End if
   vlUniqueRef:=vlUniqueRef+1
   APPEND TO LIST($hSubList;"New Item";vlUniqueRef)
   If($vbNewSubList)
      SET LIST ITEM(hList;$vlItemRef;$vsItemText;$vlItemRef;$hSublist;True)
   End if
   SELECT LIST ITEMS BY REFERENCE(hList;vlUniqueRef)
End if
```

## Example 2

See example for the **GET LIST ITEM** command.

## Example 3

See example for the **APPEND TO LIST** command.

# SET LIST ITEM FONT

SET LIST ITEM FONT ( {* ;} list ; itemRef | * ; font )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted) or Name of list type object (if * passed) |
| itemRef \| * | Longint, Operator | ⇒ | Item reference number or 0 for the last item added to the list or * for the current item of the list |
| font | String, Longint | ⇒ | Font name or number |

## Description

The **SET LIST ITEM FONT** command modifies the character font of the item specified by the *itemRef* parameter of the list whose reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

You can pass a reference number in *itemRef*. If this number does not correspond to any item of the list, the command does nothing. You can also pass 0 in *itemRef* in order to request the modification of the last item added to the list (using **APPEND TO LIST**).

Lastly, you can pass * in *itemRef*: in this case, the command will apply to the current item of the list. If several items are selected manually, the current item is the one that was selected last. If no item is selected, the command does nothing.

In the *font* parameter, pass the name or number of the font to be used. To reapply the default font of the hierarchical list, pass an empty string in *font*.

## Example

Apply the Times font to the current item of the list:

```
SET LIST ITEM FONT(*;"Mylist";*;"Times")
```

# SET LIST ITEM ICON

SET LIST ITEM ICON ( {* ;} list ; itemRef | * ; icon )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted) or Name of list type object (if * passed) |
| itemRef \| * | Longint, Operator | ⇒ | Item reference number or 0 for the last item added to the list or * for the current item of the list |
| icon | Picture | ⇒ | Icon to be associated with item |

## Description

The **SET LIST ITEM ICON** command modifies the icon associated with the item specified by the *itemRef* parameter of the list whose reference number or object name is passed in *list*.

**Note:** You can also modify the icon associated with an item using the **SET LIST ITEM PROPERTIES** command. However, this command only accepts static picture references (resource references or pictures from the picture library).

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

You can pass a reference number in *itemRef*. If this number does not correspond to an item in the list, the command does nothing. You can also pass 0 in *itemRef* to indicate the last item added to the list (using **APPEND TO LIST**).

Lastly, you can pass * in *itemRef*: in this case, the command is applied to the current item of the list. If several items are selected manually, the current item is the last one that was selected. If no item is selected, the command does nothing.

Pass a valid 4D picture expression (field, variable, pointer, etc.) in the *icon* parameter. The picture will be placed to the left of the item.

## Example

We want to assign the same picture to two different items. The following code is optimized since the picture is only loaded into memory once:

```
C_PICTURE($picture)
READ PICTURE FILE("myPict.png";$picture)
SET LIST ITEM ICON(mylist;ref1;$picture)
SET LIST ITEM ICON(mylist;ref2;$picture)
```

# SET LIST ITEM PARAMETER

SET LIST ITEM PARAMETER ( {* ;} list ; itemRef | * ; selector ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted) or Name of list type object (if * passed) |
| itemRef \| * | Operator, Longint | ⇒ | Item reference number or 0 for the last item appended to the list or * for the current list item |
| selector | String | ⇒ | Parameter constant |
| value | String, Boolean, Longint, Real | ⇒ | Value of the parameter |

## Description

The **SET LIST ITEM PARAMETER** command modifies the *selector* parameter for the *itemRef* item of the hierarchical list whose reference or object name is passed in the *list* parameter.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and the second * is passed, the syntax based on the object name is required since each representation can have its own current item.

You can pass a reference number in *itemRef*. If this number does not correspond to an item in the list, the command does nothing. You can also pass 0 in *itemRef* to indicate the last item added to the list (using **Hierarchical Lists**).

Lastly, you can pass * in *itemRef*: in this case, the command is applied to the current item of the list. If several items are selected manually, the current item is the last one that was selected. If no item is selected, the command does nothing.

In *selector*, you can pass the Additional text constant (found in the "**Hierarchical Lists**" theme) or any custom value:

- Additional text: This constant is used to add text to the right of the *itemRef* item. This additional title will always be displayed in the right part of the list, even when the user moves the horizontal scrolling cursor. When you use this constant, pass the text to be displayed in *value*.
- Custom selector: You can also pass custom text and associate it with a value of the Text, Number or Boolean type in *selector*. This value will be stored with the list item and may be retrieved using the **GET LIST ITEM PARAMETER** command. This lets you set up any type of interface associated with hierarchical lists. For example, in a list of customer names, you can store the age of each person and only display it when the corresponding item is selected.

# SET LIST ITEM PROPERTIES

SET LIST ITEM PROPERTIES ( {* ;} list ; itemRef | * ; enterable ; styles ; icon {; color} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, list is an object name (string) If omitted, list is a list reference number |
| list | ListRef, String | ⇒ | List reference number (if * omitted) or Name of list type object (if * passed) |
| itemRef \| * | Operator, Longint | ⇒ | Item reference number, or 0 for last item appended to the list, or * for the current list item |
| enterable | Boolean | ⇒ | TRUE = Enterable, FALSE = Non-enterable |
| styles | Longint | ⇒ | Font style for the item |
| icon | Longint | ⇒ | 'cicn' Mac OS-based resource ID, or 65536 + 'PICT' Mac OS-based resource ID, or 131072 + Picture Reference Number |
| color | Longint | ⇒ | RGB color value or -1 = reset to original color |

## Description

The **SET LIST ITEM PROPERTIES** command modifies the item designated by the *itemRef* parameter within the list whose reference number or object name is passed in *list*.

If you pass the first optional * parameter, you indicate that the *list* parameter is an object name (string) corresponding to a representation of the list in the form. If you do not pass this parameter, you indicate that the *list* parameter is a hierarchical list reference (*ListRef*). If you only use a single representation of the list or work with structural items (the second * is omitted), you can use either syntax. Conversely, if you use several representations of the same list and work with the current item (the second * is passed), the syntax based on the object name is required since each representation can have its own current item.

You can pass a reference number in *itemRef*. If there is no item with the item reference number that is passed, the command does nothing. You can optionally pass *0* in *itemRef* to modify the last item added to the list using **APPEND TO LIST**.

Lastly, you can pass * in *itemRef*: in this case, the command will apply to the current item of the list. If several items are selected manually, the current item is the one that was selected last. If no item is selected, the command does nothing.

If you work with item reference numbers, build a list in which items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, refer to the **Managing Hierarchical Lists** section.

**Note:** To change the text of the item or its sublist, use the command **SET LIST ITEM**.

To make an item enterable, pass TRUE in *enterable*; otherwise, pass FALSE.
**Important:** In order for an item to be enterable, it must belong to a list that is enterable. To make a whole list enterable, use the **OBJECT SET ENTERABLE** command. To make an individual list item enterable, use **SET LIST ITEM PROPERTIES**. Changing the enterable property at the list level does not affect the enterable properties of the items. However, an item can be enterable only if its list is enterable.

You specify the font style of the item in the *styles* parameter. You pass a combination (one or a sum) of the following predefined constants (**Font Styles** theme):

| Constant | Type | Value |
|---|---|---|
| Bold | Longint | 1 |
| Italic | Longint | 2 |
| Plain | Longint | 0 |
| Underline | Longint | 4 |

To associate an icon to the item, pass one of the following numeric values:

- N, where N is the resource ID of Mac OS-based 'cicn' resource
- Use PICT resource+N, where N is the the resource ID of a Mac OS-based 'PICT' resource
- Use PicRef+N, where N is the reference number of a Picture from the Design environment Picture Library

Pass zero (0), if you do not want any graphic for the item.

**Notes:**

- Use PICT resource and Use PicRef are predefined constants located in the **Hierarchical Lists** theme.

- If you want to use 4D picture expressions (fields, variables, etc.) to specify the icons of the items, use the **SET LIST ITEM ICON** command.

The *color* parameter (optional) lets you modify the color of the item text. The color must be specified in the form of an RGB color, i.e. a 4-byte longint in the 0x00RRGGBB format. For more information about this format, refer to the description of the **OBJECT SET RGB COLORS** command. Pass -1 in the *color* parameter to reset the original color of the item.

## Example 1

See the example for the **APPEND TO LIST** command.

## Example 2

The following example changes the text of the current item of *list* to bold and bright red:

```
SET LIST ITEM PROPERTIES(list;*;True;Bold;0;0x00FF0000)
```

# SET LIST PROPERTIES

SET LIST PROPERTIES ( list ; appearance {; icon {; lineHeight {; doubleClick {; multiSelections {; editable}}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ⇒ | List reference number |
| appearance | Longint | ⇒ | Graphical style of the list 1 = Hierarchical list a la Macintosh 2 = Hierarchical list a la Windows 0 = Auto appearance depending on platform |
| icon | Longint | ⇒ | 'cicn' Mac OS-based resource ID or 0 for default platform node icon |
| lineHeight | Longint | ⇒ | Minimal line height expressed in pixels |
| doubleClick | Longint | ⇒ | Expand/Collapse sublist on double-click 0 = Yes, 1= No |
| multiSelections | Longint | ⇒ | Multiple selections: 0 = No (default), 1 = Yes |
| editable | Longint | ⇒ | 0 = List is not editable by user, 1 = List is editable by user (default) |

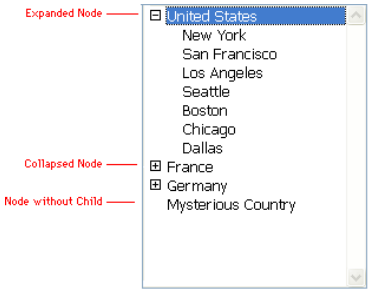
## Description

The **SET LIST PROPERTIES** command sets the appearance of the hierarchical list whose list reference you pass in *list*.

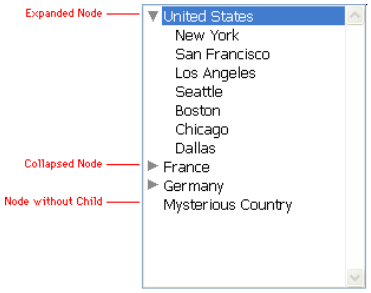
The parameter *appearance* can be one of the following predefined constants provided by 4D in the **Hierarchical Lists** theme:

| Constant | Type | Value |
|---|---|---|
| Ala Macintosh | Longint | 1 |
| Ala Windows | Longint | 2 |

In the Windows appearance, the icon (+) denotes collapsed nodes, and the icon (−) denotes expanded nodes. Nodes without child items have no icon. Here is a default hierarchical list in Windows appearance:




In the Macintosh appearance, a rightward-pointing arrow icon denotes the collapsed nodes, and a downward-pointing icon denotes the expanded nodes. Nodes without child items have no icon. Here is a default hierachical list in Macintosh appearance:




If you display a hierarchical list object without calling **SET LIST PROPERTIES** or pass 0 in the *appearence* parameter, the list appears with the default Windows or Macintosh appearances, depending on the Platform Interface property choosen for the object in the Design environment's Form Editor.

The parameter *icon* indicates the icons that will be displayed for each node. The value passed in *icon* sets the icon for collapsed nodes and *icon+1* sets the icon for expanded nodes.

For example, if you pass *15000*, the color icon *'cicn' ID=15000* will be displayed for each collapsed node and the color icon *'cicn' ID=15001* will be displayed for each expanded node.

It is therefore important to have these *'cicn'* color icon resources present in your database structure file. If a color icon resource is missing, the corresponding nodes are displayed with no icons. (You can actually take advantage of this to display a list with no icons.)

**WARNING:** When creating *'cicn'* color icon resources, use resource IDs greater than or equal to 15000. Resource IDs less than 15000 are reserved for 4D.

The resource IDs of the default Macintosh and Windows nodes are expressed by the following predefined constants provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Macintosh node | Longint | 860 |
| Windows node | Longint | 138 |

In other words, 4D provides the following *'cicn'* resources:

| ID Number | Description |
|---|---|
| 860 | Collapsed node a la Macintosh |
| 861 | Expanded node a la Macintosh |
| 138 | Collapsed node a la Windows |
| 139 | Expanded node a la Windows |

If you do not pass the parameter *icon* or pass 0, the nodes are displayed with the default icons of the chosen appearance type.

Color icon resources can be of various sizes. For example, you can create *16x16* or *32x32* color icons.

If you do not pass the parameter *lineHeight*, the line height of a hierarchical list is determined by the font and font size used for the object. If you use a color icon that is too tall or too wide, it will be displayed truncated and/or will be overridden by the text of the nodes above or below it.

Choose color icon size, font, and font size accordingly, otherwise pass in the parameter *lineHeight* the minimal line height of the hierarchical list. If the value you pass is greater than the line height derived from the font and font size used, the line height of the hierarchical list will be forced to the value you pass.

**Note: SET LIST PROPERTIES** affects the way nodes are displayed in the hierarchical list. If you would rather customize the icon of each item in the list, use the command **SET LIST ITEM PROPERTIES**.

The optional parameter *doubleClick* allows you to define that a double-click on a parent list item will not provoke the sublist to expand or to collapse. By default, a double-click on a parent list item provokes its child list to expand or to collapse. However, some user interfaces may need to deactivate this behavior. To do this, the *doubleClick* parameter should be set to 1.
Only double-click will be deactivated. Users will still be able to expand or collapse sublists by clicking on the list node.
If you omit the *doubleClick* parameter or pass 0, default behavior will be applied.

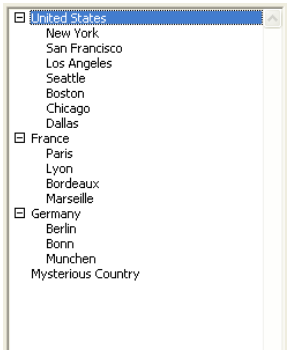
The optional *multiSelections* parameter lets you indicate whether the list must accept multiple selections.
By default, as in previous versions of 4D, you cannot simultaneously select several items of a hierarchical list. If you would like this function to be available for the list, pass the value 1 in the *multiSelections* parameter. In that case, multiple selections can be used:

- manually, using the **Shift**+**click** key combination for a continuous selection or **Ctrl**+**click** (Windows) / **Command**+**click** (Mac OS) for a discontinuous selection,
- by programming, using the **SELECT LIST ITEMS BY POSITION** and **SELECT LIST ITEMS BY REFERENCE** commands.
  If you pass 0 or omit the *multiSelections* parameter, the default behavior will be applied.

The optional *editable* parameter lets you indicate whether the list must be editable by the user when it is displayed as a choice list associated with a field or a variable during data entry. When the list is editable, a **Modify** button is added in the choice list window and the user can add, delete and sort the values through a specific editor.
If you pass 1 or omit the *editable* parameter, the list will be editable; if you pass 0, it will not be editable.

## Example

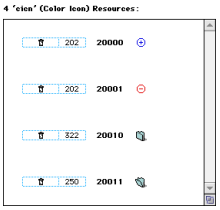
The following hierarchical list has been defined in the Design environment List Editor:




Within a form, the hierarchical list object *hlCities* reuses that list with this object method:

```
Case of
   :(Form event=On Load)
      hlCities:=Load list("Cities")
      SET LIST PROPERTIES(hlCities;vlAppearance;vlIcon)
   :(Form event=On Unload)
      CLEAR LIST(hlCities;*)
End case
```

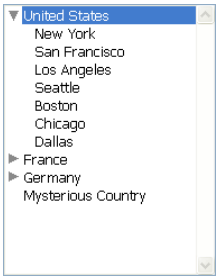
In addition, the structure file of the database has been edited so it contains the following *'cicn'* color icon resources:




1) With the following line:

```
SET LIST PROPERTIES(hlCities;Ala Macintosh;Macintosh node)
```

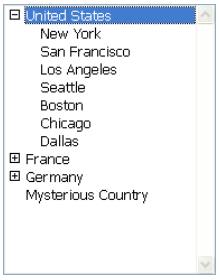
The hierarchical list will look like this:




2) With the following line:

```
SET LIST PROPERTIES(hlCities;Ala Windows;Windows node)
```

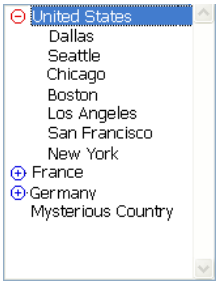
The hierarchical list will look like this:




3) With the following line:

```
SET LIST PROPERTIES(hlCities;Ala Windows;20000)
```

The hierarchical list will look like this:

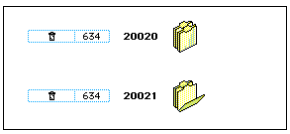



4) With the following line:

```
SET LIST PROPERTIES(hlCities;Ala Macintosh;20010)
```

The hierarchical list will look like this:


The *'cicn'* color icon resources shown are then added to the structure file of the database:




5) With the following line:

```
SET LIST PROPERTIES(hlCities;Ala Windows;20020;32)
```

The hierarchical list will look like this:

# SORT LIST

SORT LIST ( list {; > or <} )

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ⇒ | List reference number |
| > or < | Operator | ⇒ | Sorting order: > to sort in ascending order, or < to sort in descending order |

## Description

The **SORT LIST** command sorts the list whose reference number is passed in *list*.

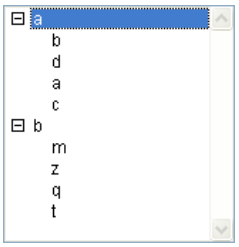
To sort in ascending order, pass >. To sort in descending order, pass <. If you omit the sorting order parameter, **SORT LIST** sorts in ascending order by default.

**SORT LIST** sorts all levels of the list; it first sorts the items of the list, then it sorts the items in each sublist (if any), and so on, through all the levels of the list. This is why you will usually apply **SORT LIST** to a list in a form. Sorting a sublist is of little interest because the order will be changed by a call to a higher level.

**SORT LIST** does not change the current list item nor the current expanded/collapsed state of the list and sublists. However, because the current item can be moved by the sorting operation, **Selected list items** may return a different position before and after the sort.

## Example

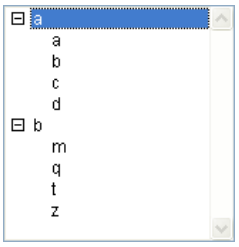
Given the list named *hList,* shown here in the Application environment:




After the execution of this code:

```
` Sort the list and it sublists in ascending order
SORT LIST(hList;>)
```

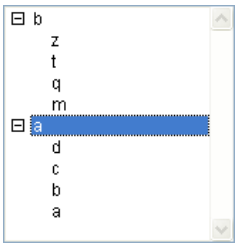
The list looks like:




After the execution of this code:

```
` Sort the list and it sublists in descending order
SORT LIST(hList;<)
```

The list looks like:



# _o_REDRAW LIST

_o_REDRAW LIST ( list )

| Parameter | Type | | Description |
|---|---|---|---|
| list | ListRef | ⇒ | List reference number |

## Description

This command serves no purpose since version 11 of 4D. All representations of hierarchical lists are now redrawn automatically.

# HTTP Client

- HTTP AUTHENTICATE
- HTTP Get
- HTTP Get certificates folder
- HTTP GET OPTION
- HTTP Request
- HTTP SET CERTIFICATES FOLDER
- HTTP SET OPTION

# HTTP AUTHENTICATE

HTTP AUTHENTICATE ( name ; password {; authMethod} {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| name | Text | ⇒ | User name |
| password | Text | ⇒ | User password |
| authMethod | Longint | ⇒ | Authentication method: 0 or omitted=not specified, 1=BASIC, 2=DIGEST |
| * | Operator | ⇒ | If passed: authentication by proxy |

## Description

The **HTTP AUTHENTICATE** command enables HTTP requests to servers requiring authentication of the client application. The BASIC and DIGEST methods are supported, as well as the presence of a proxy.

In the *name* and *password* parameters, you pass the id information required (user name and password). This information is encoded and added to the next HTTP request sent using the **HTTP Request** or **HTTP Get** command, so you need to call the **HTTP AUTHENTICATE** command before each HTTP request.

The optional *authMethod* parameter indicates the authentication method to use. You pass one of the following constants, found in the **HTTP Client** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| HTTP basic | Longint | 1 | Use BASIC authentication method |
| HTTP digest | Longint | 2 | Use DIGEST authentication method |

If you omit the *authMethod* parameter (or pass 0), you let the program choose the appropriate method to use. In this case, 4D sends an additional request in order to negotiate the authentication method.

If you pass the * parameter, this indicates that the authentication information is intended for an HTTP proxy. This setting must be implemented when there is a proxy requiring authentication between the client and the HTTP server. If the server itself is authenticated, a double authentication is necessary.

By default, authentication information is stored temporarily and reused for each request in the current process. However, it is possible to reset this information after each request using an option set by the **HTTP SET OPTION** command. In this case, you will have to execute the **HTTP AUTHENTICATE** command before each call to **HTTP Request** or **HTTP Get**.

## Example

Examples of requests with authentication:

```
 // Authentication on HTTP server in DIGEST mode
HTTP AUTHENTICATE("httpUser";"123";2)
 // Authentication on proxy in default mode
HTTP AUTHENTICATE("ProxyUser";"456";*)
$httpStatus:=HTTP Get(...)
```

# HTTP Get

HTTP Get ( url ; response {; headerNames ; headerValues}{; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| url | Text | ⇒ | URL to which to send the request |
| response | Text, BLOB, Picture, Object | ⇐ | Result of request |
| headerNames | Text array | ⇒ | Header names of the request |
| | | ⇐ | Returned header names |
| headerValues | Text array | ⇒ | Header values of the request |
| | | ⇐ | Returned header values |
| * | Operator | ⇒ | If passed, connection is maintained (keep-alive)<br>If omitted, connection is closed automatically |
| Function result | Longint | ⇒ | HTTP status code |

## Description

The **HTTP Get** command sends an HTTP GET request directly to a specific URL and processes the HTTP server response.

Pass the URL where you want the request sent in the *url* parameter. The syntax to use is:

```
http://[{user}:[{password}]@]host[:{port}][/{path}][?{queryString}]
```

For example, you can pass the following strings:

```
http://www.myserver.com
http://www.myserver.com/path
http://www.myserver.com/path?name="jones"
https://www.myserver.com/login (*)
http://123.45.67.89:8083
http://john:smith@123.45.67.89:8083
```

*(*)* During HTTPS requests, authority of the certificate is not checked.

After command execution, the *response* parameter receives the result of the request returned by the server. This result corresponds to the body of the response, with no headers.
You can pass different types of variables in *response*:

- Text: When the result is expected to be text (see note below).
- BLOB: When the result is expected to be in binary form.
- Picture: When the result is expected to be a picture.
- Object: When the result is expected to be a **C_OBJECT** object.

**Note:** When a text variable is passed in *response*, 4D will try to decode the data returned from the server. 4D first tries to retrieve the charset from the *content-type* header, then from the content using a BOM, and finally looks for any *http-equiv charset* (in html content) or *encoding* (for xml) attribute. If no charset can be detected, 4D will attempt to decode the response in ANSI. If the conversion fails, the resulting text will be empty. If you are unsure whether the server returns a charset information or a BOM, but you know the encoding, it is more accurate to pass *response* in BLOB and call **Convert to text**.

If you pass a BLOB, it contains the text, picture or any type of contents (.wav, .zip, etc.) returned by the server. You must then manage the recovery of these contents (headers are not included in the BLOB). When you pass a **C_OBJECT** type object, if the request returns a result with an "application/json" (or "*something*/json") content-type, 4D attempts to parse the JSON content in order to generate the object.

In *headerNames* and *headerValues*, you pass arrays containing the names and values of the request headers.
After this method is executed, these arrays contain the names and values of headers returned by the HTTP server. More specifically, this lets you manage cookies.

The * parameter enables the keep-alive mechanism for the server connection. By default, if this parameter is omitted, keep-alive is not enabled.

The command returns a standard HTTP status code (200=OK and so on) as returned by the server. The list of HTTP status codes is provided in RFC 2616.
If you are unable to connect to the server for a reason related to the network (DNS Failed, Server not reachable...), the

command returns 0 and an error is generated. You can intercept errors using an error-handling method installed by the **ON ERR CALL** command.

## Example 1

Retrieval of the 4D logo on the 4D Web site:

```
C_TEXT(URLPic_t)
URLPic_t:="http://www.4d.com/sites/all/themes/dimention/images/home/logo4D.jpg"
ARRAY TEXT(HeaderNames_at;0)
ARRAY TEXT(HeaderValues_at;0)
C_PICTURE(Pic_i)
$httpResponse:=HTTP Get(URLPic_t;Pic_i;HeaderNames_at;HeaderValues_at)
```

## Example 2

Retrieval of an RFC:

```
C_TEXT(URLText_t)
C_TEXT(Text_t)
URLText_t:="http://tools.ietf.org/rfc/rfc1.txt"
ARRAY TEXT(HeaderNames_at;0)
ARRAY TEXT(HeaderValues_at;0)
$httpResponse:=HTTP Get(URLText_t;Text_t;HeaderNames_at;HeaderValues_at)
```

## Example 3

Retrieval of a video:

```
C_BLOB(vBlob)
$httpResponse:=HTTP Get("http://www.example.com/video.flv";vBlob)
BLOB TO DOCUMENT("video.flv";vBlob)
```

# HTTP Get certificates folder

HTTP Get certificates folder -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Text | ➲ | Complete pathname of active certificates folder |

## Description

The **HTTP Get certificates folder** command returns the complete pathname of the active client certificates folder.

By default, 4D uses the "ClientCertificatesFolder" folder that is created next to the structure file (folder only created if necessary). However, you can create a custom folder for the current process using the **HTTP SET CERTIFICATES FOLDER** command.

## Example

You want to change certificates folder temporarily:

```
C_TEXT($certifFolder)
$certifFolder :=HTTP Get certificates folder //save current folder
HTTP SET CERTIFICATES FOLDER("C:/temp/certifTempo/")
... // execution of specific requests
HTTP SET CERTIFICATES FOLDER($certifFolder) //restore previous folder
```

# HTTP GET OPTION

HTTP GET OPTION ( option ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ⇨ | Code of option to get |
| value | Longint | ⇦ | Current value of option |

## Description

The **HTTP GET OPTION** command returns the current value of the HTTP options (options used by client for next request triggered by the **HTTP Get** or **HTTP Request** commands). The current value of an option can be the default value or it can have been modified using the **HTTP SET OPTION** command.

**Note:** The options set are local to the current process. In a component, they are local to the component being executed.

In the *option* parameter, pass the number of the option whose value you want to get. You can use one of the following predefined constants, available in the **HTTP Client** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| HTTP compression | Longint | 6 | *value* = 0 (do not compress) or 1 (compress). Default value: 0<br>This option enables or disables the compression mechanism intended to accelerate exchanges for requests between the client and server. When this mechanism is enabled, the HTTP client uses deflate or gzip compression depending on the server response. |
| HTTP display auth dial | Longint | 4 | *value* = 0 (do not display dialogue box) or 1 (display dialogue box). Default value: 0<br>This option displays the authentication dialog box when the **HTTP Get** or **HTTP Request** command is executed. By default, this command never displays the dialog box and you must normally use the **HTTP AUTHENTICATE** command. However, if you want an authentication dialog box to appear so that users can enter their identifiers, then pass 1 in value. The dialog box only appears when the request requires authentication. |
| HTTP follow redirect | Longint | 2 | *value* = 0 (do not accept redirections) or 1 (accept redirections).<br>Default value = 1 |
| HTTP max redirect | Longint | 3 | *value* = Maximum number of redirections accepted<br>Default value = 2 |
| HTTP reset auth settings | Longint | 5 | *value* = 0 (do not delete information) or 1 (delete information). Default value: 0<br>This option indicates to 4D to reset the authentication information of the user (user name, password, method) after each execution of the **HTTP Get** or **HTTP Request** command in the same process. By default, this information is kept and reused for each request. Pass 1 in *value* to delete this information after each call. Note that regardless of the setting, this information is deleted when the process is killed. |
| HTTP timeout | Longint | 1 | *value* = timeout of client request, expressed in seconds. This timeout sets how long the HTTP client waits for the server to respond. After this period of time has passed, the client closes the session and the request is lost.<br>By default, this timeout is 120 seconds. It can be changed because of specific characteristics (network state, request characteristics, etc.). |

In the *value* parameter, pass a variable to receive the current value of the *option*.

# HTTP Request

HTTP Request ( httpMethod ; url ; contents ; response {; headerNames ; headerValues}{; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| httpMethod | Text | ➡ | HTTP method for request |
| url | Text | ➡ | URL to which to send the request |
| contents | Text, BLOB, Picture, Object | ➡ | Contents of request body |
| response | Text, BLOB, Picture, Object | ⬅ | Result of request |
| headerNames | Text array | ➡ | Header names of the request |
| | | ⬅ | Returned header names |
| headerValues | Text array | ➡ | Header values of the request |
| | | ⬅ | Returned header values |
| * | Operator | ➡ | If passed, connection is maintained (keep-alive)<br>If omitted, connection is closed automatically |
| Function result | Longint | ➲ | HTTP status code |

## Description

The **HTTP Request** command enables all types of HTTP requests to be sent to a specific URL and processes the HTTP server response.

Pass the HTTP method of the request in the *httpMethod* parameter. You can use one of the following constants, found in the **HTTP Client** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| HTTP DELETE method | String | DELETE | See RFC 2616 |
| HTTP GET method | String | GET | See RFC 2616. Same as using **HTTP Get** command. |
| HTTP HEAD method | String | HEAD | See RFC 2616 |
| HTTP OPTIONS method | String | OPTIONS | See RFC 2616 |
| HTTP POST method | String | POST | See RFC 2616 |
| HTTP PUT method | String | PUT | See RFC 2616 |
| HTTP TRACE method | String | TRACE | See RFC 2616 |

Pass the URL where you want the request sent in the *url* parameter. The syntax to use is:

```
http://[{user}:[{password}]@]host[:{port}][/{path}][?{queryString}]
```

For example, you can pass the following strings:

```
http://www.myserver.com
http://www.myserver.com/path
http://www.myserver.com/path?name="jones"
https://www.myserver.com/login (*)
http://123.45.67.89:8083
http://john:smith@123.45.67.89:8083
```

*(*)* During HTTPS requests, authority of the certificate is not checked.

Pass the body of the request in the *contents* parameter. Data passed in this parameter depends on the HTTP method of the request.
You can send data of the Text, BLOB, Picture or Object type. When the *content-type* is not specified, the following types are used:

- for text: text/plain - UTF8
- for BLOBs: application/byte-stream
- for pictures: known MIME type (best for Web).
- for **C_OBJECT** objects: application/json

After command execution, the *response* parameter receives the result of the request returned by the server. This result corresponds to the body of the response, with no headers.
You can pass different types of variables in *response*:

- Text: When the result is expected to be text (see note below).
- BLOB: When the result is expected to be in binary
- Picture: When the result is expected to be a picture.
- **C_OBJECT** Object: When the result is expected to be an object.

**Note:** When a text variable is passed in *response*, 4D will try to decode the data returned from the server. 4D first tries to retrieve the charset from the *content-type* header, then from the content using a BOM, and finally looks for any *http-equiv charset* (in html content) or *encoding* (for xml) attribute. If no charset can be detected, 4D will attempt to decode the response in ANSI. If the conversion fails, the resulting text will be empty. If you are unsure whether the server returns a charset information or a BOM, but you know the encoding, it is more accurate to pass *response* in BLOB and call **Convert to text**.

When you pass a **C_OBJECT** type variable in the *response* parameter, if the request returns a result with an "application/json" (or "*something*/json") content-type, 4D attempts to parse the JSON content in order to generate the object.

If the result returned by the server does not correspond to the *response* variable type, it is left blank and the OK system variable is set to 0.

In *headerNames* and *headerValues*, you pass arrays containing the names and values of the request headers.
After this method is executed, these arrays contain the names and values of headers returned by the HTTP server. More specifically, this lets you manage cookies.

The * parameter enables the keep-alive mechanism for the server connection. By default, if this parameter is omitted, keep-alive is not enabled.

The command returns a standard HTTP status code (200=OK and so on) as returned by the server. The list of HTTP status codes is provided in RFC 2616.
If you are unable to connect to the server for a reason related to the network (DNS Failed, Server not reachable...), the command returns 0 and an error is generated. You can intercept errors using an error-handling method installed by the **ON ERR CALL** command.

## Example 1

Requesting for a record deletion from a remote database:

```
C_TEXT($response)
$body_t:="{record_id:25}"
$httpStatus_l:=HTTP Request(HTTP_DELETE_method;"database.example.com";$body_t;$response)
```

**Note:** You have to process the request appropriately on the remote server, **HTTP Request** only handles the request and the returned result.

## Example 2

Requesting to add a record to a remote database:

```
C_TEXT($response)
$body_t:="{fName:'john',fName:'Doe'}"
$httpStatus_l:=HTTP Request(HTTP_PUT_method;"database.example.com";$body_t;$response)
```

**Note:** You have to process the request appropriately on the remote server, **HTTP Request** only handles the request and the returned result.

## Example 3

Request to add a record in JSON to a remote database::

```
C_OBJECT($content)
OB SET($content;"lastname";"Doe";"firstname";"John")
$result:=HTTP Request(HTTP_PUT_method;"database.example.com";$content;$response)
```

# HTTP SET CERTIFICATES FOLDER

HTTP SET CERTIFICATES FOLDER ( certificatesFolder )

| Parameter | Type | | Description |
|---|---|---|---|
| certificatesFolder | Text | ⇒ | Pathname and name of client certificates folder |

## Description

The **HTTP SET CERTIFICATES FOLDER** command modifies the active client certificates folder for all processes of the current session.

The client certificates folder is the one where 4D looks for the client certificate files that are required by Web servers. By default, as long as the **HTTP SET CERTIFICATES FOLDER** command is not executed, 4D uses a folder named "ClientCertificatesFolder" that is created next to the structure file. This folder is only created when necessary.

In 4D v14, it is now possible to use several client certificates.

In *certificatesFolder*, pass the pathname of the custom folder containing the client certificates. You can pass either a pathname relative to the application structure file, or an absolute pathname. The path must be expressed using the system syntax, for example:

- (OS X): Disk:Applications:myserv:folder
- (Windows): C:¥Applications¥myserv¥folder

Once this command has been executed, the new path is immediately taken into account by commands such as **HTTP Request** that are executed later on (you do not have to restart the application). It is used in all the processes of the database.

If the folder specified does not exist at the location defined, or if the pathname passed in *certificatesFolder* is not valid, an error is generated. You can intercept this error using an error-handling method installed by the **ON ERR CALL** command.

## About SSL certificates

As described in the **Using TLS Protocol** section, SSL certificates managed by 4D must be in the **PEM format**. If your certificate provider (for example, startssl) sends you a certificate that is in a binary format such as .crt, .pfx or .p12 (the format also depends on your browser), you have to convert it to PEM format in order to be able to use it. There are Web sites such as sslshopper where you can do this conversion on-line.

## Example

You want to change certificates folder temporarily:

```
C_TEXT($certifFolder)
$certifFolder :=HTTP Get certificates folder //save current folder
HTTP SET CERTIFICATES FOLDER("C:/temp/certifTempo/")
... // execution of specific requests
HTTP SET CERTIFICATES FOLDER($certifFolder) //restore previous folder
```

# HTTP SET OPTION

HTTP SET OPTION ( option ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ⇒ | Code of option to set |
| value | Longint | ⇒ | Value of option |

## Description

The **HTTP SET OPTION** command sets different options that are used during the next HTTP request triggered by the **HTTP Get** or **HTTP Request** commands. You call this command as many times as there are options to set.

**Note:** Options set are local to the current process. For components, they are local to the component being executed. .

In the *option* parameter, pass the number of the option to be set and in the *value* parameter, pass the new value of this option. For the *option* parameter, you can use one of the following predefined constants, available in the **HTTP Client** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| HTTP compression | Longint | 6 | *value* = 0 (do not compress) or 1 (compress). Default value: 0<br>This option enables or disables the compression mechanism intended to accelerate exchanges for requests between the client and server. When this mechanism is enabled, the HTTP client uses deflate or gzip compression depending on the server response. |
| HTTP display auth dial | Longint | 4 | *value* = 0 (do not display dialogue box) or 1 (display dialogue box). Default value: 0<br>This option displays the authentication dialog box when the **HTTP Get** or **HTTP Request** command is executed. By default, this command never displays the dialog box and you must normally use the **HTTP AUTHENTICATE** command. However, if you want an authentication dialog box to appear so that users can enter their identifiers, then pass 1 in value. The dialog box only appears when the request requires authentication. |
| HTTP follow redirect | Longint | 2 | *value* = 0 (do not accept redirections) or 1 (accept redirections).<br>Default value = 1 |
| HTTP max redirect | Longint | 3 | *value* = Maximum number of redirections accepted<br>Default value = 2 |
| HTTP reset auth settings | Longint | 5 | *value* = 0 (do not delete information) or 1 (delete information). Default value: 0<br>This option indicates to 4D to reset the authentication information of the user (user name, password, method) after each execution of the **HTTP Get** or **HTTP Request** command in the same process. By default, this information is kept and reused for each request. Pass 1 in *value* to delete this information after each call. Note that regardless of the setting, this information is deleted when the process is killed. |
| HTTP timeout | Longint | 1 | *value* = timeout of client request, expressed in seconds. This timeout sets how long the HTTP client waits for the server to respond. After this period of time has passed, the client closes the session and the request is lost.<br>By default, this timeout is 120 seconds. It can be changed because of specific characteristics (network state, request characteristics, etc.). |

You can call options in any order. If the same option is set more than once, only the value of the last call is taken into account.

# Import and Export

- EXPORT DATA
- EXPORT DIF
- EXPORT ODBC
- EXPORT SYLK
- EXPORT TEXT
- IMPORT DATA
- IMPORT DIF
- IMPORT ODBC
- IMPORT SYLK
- IMPORT TEXT

# EXPORT DATA

EXPORT DATA ( fileName {; project {; *}} )

| Parameter | Type | | Description |
|---|---|---|---|
| fileName | String | ⇒ | Full path name of the export file |
| project | Text variable, BLOB variable | ⇒ | Contents of the export project |
| | | ⇐ | New contents of the export project (if the * parameter has been passed) |
| * | Operator | ⇒ | Displays the export dialog box and updates the project |

## Description

The **EXPORT DATA** command exports data in the *fileName* file.4D can export data in the following formats: Text, Fixed length text, XML, SYLK, DIF, DBF (dBase), and 4D.

If you pass an empty string in *fileName*, **EXPORT DATA** displays the standard save file dialog box, allowing the user to define the name, type, and location of the export file. Once the dialog box has been accepted, the **Document** system variable contains the access path and the name of the file. If the user clicks **Cancel**, the execution of the command is stopped and the **OK** system variable is equal to 0.

The optional *project* parameter lets you use a project to export data. When you pass this parameter, the export is carried out directly, without any user intervention (unless you use the * option, see below). If you don't pass this parameter, the export dialog box is displayed. The user can define the export parameters or load an existing export project.

An export project contains all the export parameters such as the tables and fields to export, delimiters, etc. In the *project* parameter, you can pass either a Text variable containing XML or a Text variable containing a reference to a pre-existing DOM element, or a BLOB. Projects may have been created by programming (XML format projects only) or by loading parameters that were previously defined in the export dialog box.In the latter case, you have two solutions available:

- Use the **EXPORT DATA** command with an empty *project* parameter and the optional * parameter, then store the resulting *project* parameter in a Text or BLOB type field (see below). This solution allows you to save the project with the data file.
- Save the project to disk, then load it for example by using the **DOM Parse XML source** command, and pass its reference in the *project* parameter

**Compatibility note:** Beginning with version 12 of 4D, export projects are encoded in XML. 4D can open export projects generated with previous 4D versions (BLOB format) but any projects created starting with v12 can no longer be opened with v11 or earlier versions. We now recommend that you use Text variables for handling export files.

The optional parameter *, if it is specified, forces the display of the export dialog box with the parameters defined in *project*. This feature allows you to use a predefined project, while still having the possibility to modify one or more of the parameters. Furthermore, the project parameter contains, after closing the export dialog box, the parameters of the "new" project. You can then store the new project in a Text field, on disk, etc.

If the export was successful, the **OK** system variable is set to 1.

## Example 1

This example illustrates the use of the **EXPORT DATA** command to export data in binary format.

- This method makes a loop on all the database tables and calls the ***ExportBinary*** method:

```
C_TEXT($ExportPath)
C_LONGINT($i)
$ExportPath:=Select folder("Please select the export folder:")
If(Ok=1)
   For($i;1;Get last table number
      If(Is table number valid($i))
         ExportBinary(Table($i);$ExportPath+Table name($i);True)
      End if
   End for
End if
```

- Here is the code for the ***ExportBinary*** method:

```
C_POINTER($1) //table
C_TEXT($2) //path of destination file
C_BOOLEAN($3) //export all records
C_LONGINT($i)
C_TEXT($ref)
$ref:=DOM Create XML Ref("settings-import-export")
 // Export the table "$1" in '4D' binary format, all the records or only the current selection
DOM SET XML ATTRIBUTE($ref;"table_no";Table($1);"format";"4D";"all_records";$3)
 // Definition of fields to export
For($i;1;Get last field number($1))
   If(Is field number valid($1;$i))
      $elt:=DOM Create XML element($ref;"field";"table_no";Table($1);"field_no";$i)
   End if
End for
EXPORT DATA($2;$ref)
If(Ok=0)
   ALERT("Error during export of table "+Table name($1))
End if
DOM CLOSE XML($ref)
```

## Example 2

This example creates an empty project and stores the parameters set by the user in the export dialog box there:

```
C_TEXT($exportParams)
EXPORT DATA("DocExport.txt";$exportParams;*) //Display of the export dialog box
```

## System variables and sets

If the user clicks **Cancel** in the standard open file dialog box or in the export dialog box, the OK system variable is equal to 0. If the export was successful, the OK system variable is equal to 1.

# EXPORT DIF

EXPORT DIF ( {aTable ;} document )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to export data,or Default table, if omitted |
| document | String | ⇒ | DIF document to receive the data |

## Description

The **EXPORT DIF** command writes data from the records of the current selection of *aTable* in the current process. The data is written to *document*, a Windows or Macintosh DIF document on the disk.

The export operation is performed through the current output form. The export operation writes fields and variables based on the entry order of the output form. For this reason, you should use an output form that contains only the fields or enterable objects that you wish to export. Do not place buttons or other extraneous objects on the export form.Subform objects are ignored.

An On Load event is sent to the form method for each record that is exported. Use this event to set the variables you may use in the export form.

The *document* parameter can name a new or existing document. If *document* is given the same name as an existing document, the existing document is overwritten. The *document* can include a path that contains volume and folder names. If you pass an empty string, the standard Save File dialog box is displayed. If the user cancels this dialog, the export operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during export. The user can cancel the operation by clicking a Stop button. If the export is successfully completed, the OK system variable is set to 1. If the operation is canceled or an error occurs, the OK system variable is set to 0. The thermometer can be hidden with the **MESSAGES OFF** command.

By default, the command uses the UTF-8 character set. Since DIF format documents generally use the IBM437 character set, you may need to use the **USE CHARACTER SET** command to specify the appropriate character set.

When using **EXPORT DIF**, the default field delimiter is the tab character (code 9). The default record delimiter is the carriage return (code 13). You can modify these values by assigning new values to the two **System Variables** *FldDelimit* and *RecDelimit*. The user can change these default values in the export dialog box of the Design mode. Since Text fields can contain carriage returns, be careful if you use the carriage return as the field delimiter for fields to be exported.

## Example

The following example exports data to a DIF document. The method first sets the output form so that the data will be exported through the correct form, then performs the export:

```
FORM SET OUTPUT([People];"Export")
EXPORT DIF([People];"NewPeople.dif"")  ` Export to the "NewPeople.dif" document
```

## System variables and sets

OK is set to 1 if the export is successfully completed; otherwise, it is set to 0.

# EXPORT ODBC

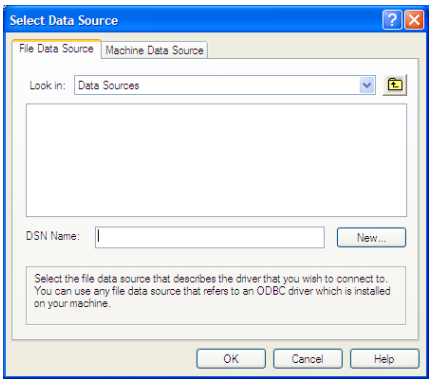
EXPORT ODBC ( sourceTable {; project {; *}} )

| Parameter | Type | | Description |
|---|---|---|---|
| sourceTable | String | ⇒ | Name of table in ODBC data source |
| project | BLOB | ⇒ | Contents of export project |
| | | ⇐ | New contents of export project (if * is passed) |
| * | Operator | ⇒ | Display of export dialog box and project update |

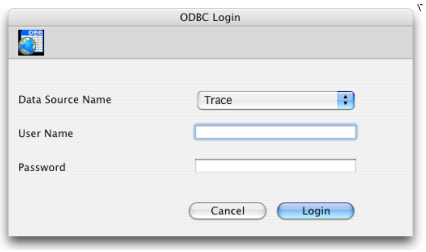
## Description

The **EXPORT ODBC** command is used to export data in the *sourceTable* table of an external ODBC source.
If you call the **EXPORT ODBC** command outside of a connection opened previously using the **SQL LOGIN** command, the ODBC data source selection dialog box is displayed:

**Windows**




**Mac OS**




If the user clicks **Cancel** in this dialog box, execution is stopped and the system variable OK is set to 0.

**Note:** This command cannot be used in the case of connections with the internal SQL kernel of 4D.

If you do not pass the optional *project* parameter, 4D displays the ODBC export dialog box, which allows the user to configure the operation. For more information about this dialog box, refer to the Design Reference manual.

If you pass a BLOB containing a valid ODBC export project in the *project* parameter, the export will be carried out directly, without any user intervention. To do this, you simply need to load a project that has been saved on disk beforehand into the field or the BLOB variable that you pass in the *project* parameter, using the **DOCUMENT TO BLOB** command. ODBC export projects are saved via the ODBC export dialog box.

You can also use the **EXPORT ODBC** command with an empty *project* parameter and the optional * parameter, then store the *project* parameter in a BLOB field (see below). On the one hand, this solution lets you store the project with the data file and, on the other, to avoid the phase of loading it from the disk into a BLOB.

**Note:** Refer to the **EXPORT DATA** command for an example concerning the definition of an empty project. Please note that projects generated in the ODBC export dialog box are not compatible with the commands or the standard export dialog box of 4D.

The optional * parameter, if it is set, displays the ODBC data export dialog box with the settings defined in *project* (if any). This allows you to use a predefined project while still being able to modify one or more parameters. Moreover, in this case, the *project* parameter contains the parameters of the “new” project after the dialog box is closed. You can then store it in a BLOB field, in a file on disk, etc.

## System variables and sets

If the user clicks **Cancel** in either of the two dialog boxes (for selecting the data source or the export settings), the system variable OK is set to 0. If the export is carried out correctly, the system variable OK is set to 1.

# EXPORT SYLK

EXPORT SYLK ( {aTable ;} document )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to export data, or Default table, if omitted |
| document | String | ⇒ | SYLK document to receive the data |

## Description

The **EXPORT SYLK** command writes data from the records of the current selection of *aTable* in the current process. The data is written to *document*, a Windows or Macintosh Sylk document on the disk.

The export operation is performed through the current output form. The export operation writes fields and variables based on the entry order of the output form. For this reason, you should use an output form that contains only the fields or enterable objects that you wish to export. Do not place buttons or other extraneous objects on the export form.Subform objects are ignored.

An On Load event is sent to the form method for each record that is exported. Use this event to set the variables you may use in the export form.

The *document* parameter can name a new or existing document. If *document* is given the same name as an existing document, the existing document is overwritten. The *document* can include a path that contains volume and folder names. If you pass an empty string, the standard Save File dialog box is displayed. If the user cancels this dialog, the export operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during export. The user can cancel the operation by clicking a Stop button. If the export is successfully completed, the OK system variable is set to 1. If the operation is canceled or an error occurs, the OK system variable is set to 0. The thermometer can be hidden with the **MESSAGES OFF** command.

By default, the command uses the UTF-8 character set. Since SYLK format documents generally use the ISO-8859-1 character set, you may need to use the **USE CHARACTER SET** command to specify the appropriate character set.

When using **EXPORT SYLK**, the default field delimiter is the tab character (code 9). The default record delimiter is the carriage return (code 13) under OS X and the carriage return+line feed (code 13 + code 10) under Windows. You can modify these values by assigning new values to the two **System Variables** *FldDelimit* and *RecDelimit*. The user can change these default values in the export dialog box of the Design mode. Note that if exported fields contain characters defined as field or record delimiters, these characters are automatically replaced with spaces in the exported file, in order to avoid disrupting the importing process.

## Example

The following example exports data to a SYLK document. The method first sets the output form so that the data will be exported through the correct form, then performs the export:

```
FORM SET OUTPUT([People];"Export")
EXPORT SYLK([People];"NewPeople.slk") ` Export to the "NewPeople.slk" document
```

## System variables and sets

OK is set to 1 if the export is successfully completed; otherwise, it is set to 0.

# EXPORT TEXT

EXPORT TEXT ( {aTable ;} document )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to export data, or Default table, if omitted |
| document | String | ⇒ | Text document to receive the data |

## Description

The **EXPORT TEXT** command writes data from the records of the current selection of *aTable* in the current process. The data is written to *document*, a Windows or Macintosh text document on the disk.

The export operation is performed through the current output form. The export operation writes fields and variables based on the entry order of the output form. For this reason, use an output form that contains only the fields or enterable objects that you wish to export. Do not place buttons or other extraneous objects on the export form.Subform objects are ignored.

An On Load event is sent to the form method for each record that is exported. Use this event to set the variables you may use in the export form.

The *document* parameter can name a new or existing document. If *document* is given the same name as an existing document, the existing document is overwritten. The *document* can include a path that contains volume and folder names. If you pass an empty string, the standard Save File dialog box is displayed. If the user cancels this dialog, the export operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during export. The user can cancel the operation by clicking a Stop button. If the export is successfully completed, the OK system variable is set to 1. If the operation is canceled or an error occurs, the OK system variable is set to 0. The thermometer can be hidden with the **MESSAGES OFF** command.

By default, the command uses by default the UTF-8 character set. You can use the **USE CHARACTER SET** command to change this character set.

Using **EXPORT TEXT**, the default field delimiter is the tab character (code 9). The default record delimiter is the carriage return character (code 13) under OS X and the carraige return+line feed (code 13 + code 10) under Windows. You can change these defaults by assigning values to the two delimiter **System Variables**: **FldDelimit** and **RecDelimit**. The user can change the default values in the Design environment Export Data dialog box. Note that if exported fields contain characters defined as field or record delimiters, these characters are automatically replaced with spaces in the exported file, in order to avoid disrupting the importing process.

## Example

This example exports data to a text document. The method first sets the output form so that the data will be exported through the correct form, changes the 4D delimiter variables, then performs the export:

```
FORM SET OUTPUT([People];"Export")
FldDelimit:=27 ` Set field delimiter to Escape character
RecDelimit:=10 ` Set record delimiter to Line Feed character
EXPORT TEXT([People];"NewPeople.txt") ` Export to the "NewPeople.txt" document
```

## System variables and sets

OK is set to 1 if the export is successfully completed; otherwise, it is set to 0.

# IMPORT DATA

IMPORT DATA ( fileName {; project {; *}} )

| Parameter | Type | | Description |
|---|---|---|---|
| fileName | String | ⇒ | Access path and name of the import file |
| project | Text variable, BLOB variable | ⇒ | Contents of the import project |
| | | ⇐ | New contents of the import project (if the * parameter has been passed) |
| * | Operator | ⇒ | Displays the import dialog box and updates the project |

## Description

The **IMPORT DATA** command imports the data located in the *fileName* file. 4D can import the data in the following formats: Text, Fixed length text, XML, SYLK, DIF, DBF (dBase), and 4D.

If you pass an empty string to *fileName*, **IMPORT DATA** displays the standard save file dialog box, allowing the user to define the name, type, and location of the import file. Once the dialog box has been accepted, the *Document* system variable contains the access path and the name of the file. If the user clicks **Cancel**, the execution of the command is stopped and the *OK* system variable is set to 0.

The optional *project* parameter lets you use a project to import data. When you pass this parameter, the import is carried out directly, without any user intervention (unless you use the * option, see below). If you don't pass this parameter, the import dialog box is displayed. The user can define the import parameters or load an existing import project.

An import project contains all the import parameters such as the tables and fields in which to import, the delimiters to use, and so on. In the *project* parameter, you can pass either a Text variable containing XML or a Text variable containing a reference to a pre-existing DOM element, or a BLOB. Projects may have been created by programming (XML format projects only) or by loading parameters that were previously defined in the import dialog box.In the latter case, you have two solutions available:

- Use the **IMPORT DATA** command with an empty project parameter and the optional parameter *, then store the resulting *project* parameter in a Text or BLOB field (see below). This solution allows you to save the project with the data file.
- Save the project to disk, then load it for example using the **DOM Parse XML source** command, and pass its reference in the *project* parameter.

**Compatibility note:** Beginning with version 12 of 4D, import projects are encoded in XML. 4D can open import projects generated with previous 4D versions (BLOB format) but any projects created starting with v12 can no longer be opened with v11 or earlier versions. We now recommend that you use Text variables for handling import files.

The optional * parameter, if it is specified, forces the display of the import dialog box with the import parameters set as they were defined in *project*. This feature allows you to use a predefined project, while still having the possibility to modify one or more of the parameters. Furthermore, the *project* parameter contains, after closing the import dialog box, the parameters of the "new" project. You can then store the new project in a BLOB field, on disk, and so on.

If the import was successful, the *OK* system variable is set to 1.

**Note:** Refer to the **EXPORT DATA** command for an example concerning the definition of an empty project.

## System variables and sets

If the user clicks **Cancel** in the standard save file dialog box or in the import dialog box, the OK system variable is set to 0. If the import was successful, the OK system variable is set to 1.

# IMPORT DIF

IMPORT DIF ( {aTable ;} document )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table into which to import data, or Default table, if omitted |
| document | String | ⇒ | DIF document from which to import data |

## Description

The **IMPORT DIF** command reads data from *document*, a Windows or Macintosh DIF document, into the table *aTable* by creating new records for that table.

The import operation is performed through the current input form. The import operation reads fields and variables based on the layering of objects in the input form. For this reason, you should be very careful about the front-to-back order of text objects (fields and variables) in the form. The first object into which data will be imported should be in the back of the form, and so on. If the number of fields or variables in the form does not match the number of fields being imported, the extra ones are ignored. An input form used for importing cannot contain any buttons. Subform objects are ignored.

**Note:** One way to ensure that the data is imported into the correct objects is to select the object into which the first field should be imported and move it to the front. Continue to move the fields and variables to the front, in order, making sure that you have one field or variable for each field being imported.

An On Validate event is sent to the form method for each record that is imported. Use this event to copy data from variables to fields, if you use variables in the import form.

The *document* parameter can include a path that contains volume and folder names. If you pass an empty string, the standard Open File dialog box is displayed. If the user cancels this dialog, the import operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during import. The user can cancel the operation by clicking a Stop button. Records that have already been imported will not be removed if the user presses the Stop button. If the import is successfully completed, the OK system variable is set to 1. If an error occurs or the operation was interrupted, the OK variable is set to 0. The thermometer can be hidden with the **MESSAGES OFF** command.

By default, the command uses the UTF-8 character set. Since DIF format documents generally use the IBM437 character set, you may need to use the **USE CHARACTER SET** command to specify the appropriate character set.

When using **IMPORT DIF**, the default field delimiter is the tab character (code 9). The default record delimiter is the carriage return (code 13). You can modify these values by assigning new values to the two **System Variables** *FldDelimit* and *RecDelimit*. The user can change these default values in the export dialog box of the Design mode. Since Text fields can contain carriage returns, be careful if you use the carriage return as the field delimiter for fields to be exported.

## Example

The following example imports data from a DIF document. The method first sets the input form so that the data will be imported through the correct form, then performs the import:

```
FORM SET INPUT([People];"Import")
IMPORT DIF([People];"NewPeople.dif") ` Import from "NewPeople.dif" document
```

## System variables and sets

OK is set to 1 if the import is successfully completed; otherwise, it is set to 0.

# IMPORT ODBC

IMPORT ODBC ( sourceTable {; project {; *}} )

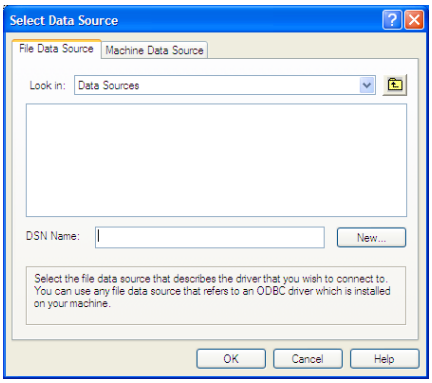
| Parameter | Type | | Description |
|---|---|---|---|
| sourceTable | String | ⇒ | Name of table in ODBC data source |
| project | BLOB | ⇒ | Contents of import project |
| | | ⇐ | New contents of import project (if * is passed) |
| * | Operator | ⇒ | Display of import dialog box and project update |

## Description

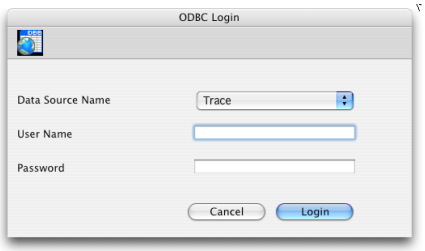
The **IMPORT ODBC** command is used to import data from the *sourceTable* table of an external ODBC source.

If you call the **IMPORT ODBC** command outside of a connection previously opened using the **SQL LOGIN**, command, the ODBC data source selection dialog box is displayed:

**Windows**




**Mac OS**




If the user clicks **Cancel** in the dialog box, execution is stopped and the system variable OK is set to 0.

**Note:** This command cannot be used in the case of connections with the internal SQL kernel of 4D.

If you do not pass the optional *project* parameter, 4D displays the ODBC import dialog box which allows the user to configure the operation. For more information about this dialog box, refer to the Design Reference manual.
If you pass a BLOB containing a valid ODBC import project in the *project* parameter, the import will be carried out directly, without any user intervention. To do this, you simply need to load a project that has been saved on disk beforehand into the field or the BLOB variable that you pass in the *project* parameter, using the **DOCUMENT TO BLOB** command. ODBC import projects are saved via the ODBC import dialog box.

You can also use the **IMPORT ODBC** command with an empty *project* parameter and the optional * parameter, then store the *project* parameter in a BLOB field (see below). On the one hand, this solution lets you store the project with the data file and, on the other, to avoid the phase of loading it from the disk into a BLOB.

**Note:** Refer to the **EXPORT DATA** command for an example concerning the definition of an empty project. Please note that projects generated in the ODBC import dialog box are not compatible with the commands or the standard import dialog box of 4D.

The optional * parameter, if it is set, displays the ODBC data import dialog box with the settings defined in *project* (if any). This allows you to use a predefined project while still being able to modify one or more parameters. Moreover, in this case, the *project* parameter contains the parameters of the “new” project after the dialog box is closed. You can then store it in a BLOB field, in a file on disk, etc.

## System variables and sets

If the user clicks **Cancel** in either of the two dialog boxes (for selecting the data source or the import settings), the system variable OK is set to 0. If the import is carried out correctly, the system variable OK is set to 1.

# IMPORT SYLK

IMPORT SYLK ( {aTable ;} document )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table into which to import data, or Default table, if omitted |
| document | String | ⇒ | SYLK document from which to import data |

## Description

The **IMPORT SYLK** command reads data from *document*, a Windows or Macintosh SYLK document, into the table *aTable* by creating new records for that table.

The import operation is performed through the current input form. The import operation reads fields and variables based on the layering of objects in the input form. For this reason, you should be very careful about the front-to-back order of text objects (fields and variables) in the form. The first object into which data will be imported should be in the back of the form, and so on. If the number of fields or variables in the form does not match the number of fields being imported, the extra ones are ignored. An input form used for importing cannot contain any buttons. Subform objects are ignored.

**Note:** One way to ensure that the data is imported into the correct objects is to select the object into which the first field should be imported and move it to the front. Continue to move the fields and variables to the front, in order, making sure that you have one field or variable for each field being imported.

An On Validate event is sent to the form method for each record that is imported. If you use variables in the import form, use this event to copy data from variables to fields.

The *document* parameter can include a path that contains volume and folder names. If you pass an empty string, the standard Open File dialog box is displayed. If the user cancels this dialog, the import operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during the import. The user can cancel the operation by clicking a Stop button. Records that have already been imported will not be removed if the user presses the Stop button. If the import is successfully completed, the OK system variable is set to 1. If an error occurs or the operation was interrupted, the OK variable is set to 0. The thermometer can be hidden with the **MESSAGES OFF** command.

By default, the command uses the UTF-8 character set. Since SYLK format documents generally use the ISO-8859-1 character set, you may need to use the **USE CHARACTER SET** command to specify the appropriate character set.

When using **IMPORT SYLK**, the default field delimiter is the tab character (code 9). The default record delimiter is the carriage return (code 13). You can modify these values by assigning new values to the two **System Variables** *FldDelimit* and *RecDelimit*. The user can change these default values in the export dialog box of the Design mode. Since Text fields can contain carriage returns, be careful if you use the carriage return as the field delimiter for fields to be exported.

## Example

The following example imports data from a SYLK document. The method first sets the input form so the data will be imported through the correct form, then performs the import:

```
FORM SET INPUT([People];"Import")
IMPORT SYLK([People];"NewPeople.slk") ` Import from "NewPeople.slk" document
```

## System variables and sets

OK is set to 1 if the import is successfully complete; otherwise, it is set to 0.

# IMPORT TEXT

IMPORT TEXT ( {aTable ;} document )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table into which to import data, or Default table, if omitted |
| document | String | ⇒ | Text document from which to import data |

## Description

The **IMPORT TEXT** command reads data from *document*, a Windows or Macintosh text document, into the table *aTable* by creating new records for that table.

The import operation is performed through the current input form. The import operation reads fields and variables based on the layering of objects in the input form. For this reason, you should be very careful about the front-to-back order of text objects (fields and variables) in the form. The first object into which data will be imported should be in the back of the form, and so on. If the number of fields or variables in the form does not match the number of fields being imported, the extra ones are ignored. An input form used for importing cannot contain any buttons. Subform objects are ignored.

**Note:** One way to ensure that the data is imported into the correct objects is to select the object into which the first field should be imported and move it to the front. Continue to move fields and variables to the front in order, making sure that you have one field or variable for each field being imported.

An On Validate event is sent to the form method for each record that is imported. If you use variables in the import form, use this event to copy data from variables to fields.

The *document* parameter can include a path that contains volume and folder names. If you pass an empty string, the standard Open File dialog box is displayed. If the user cancels this dialog, the import operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during import. The user can cancel the operation by clicking a button labeled Stop. Records that have already been imported will not be removed if the user presses the Stop button. If the import is successfully completed, the OK system variable is set to 1. If an error occurs or the operation was interrupted, the OK variable is set to 0. The thermometer can be hidden with the **MESSAGES OFF** command.

By default, the command uses the UTF-8 character set. You can use the **USE CHARACTER SET** command to change this character set.

Using **IMPORT TEXT**, the default field delimiter is the tab character (code 9). The default record delimiter is the carriage return character (code 13). You can change these defaults by assigning values to the two delimiter **System Variables**: *FldDelimit* and *RecDelimit*. The user can change the defaults in the Design environment's Import Data dialog box. Text fields may contain carriage returns, therefore, be careful when using a carriage return as a delimiter if you are importing text fields.

## Example

The following example imports data from a text document. The method first sets the input form so that the data will be imported through the correct form, changes the 4D delimiter variables, then performs the import:

```
FORM SET INPUT([People];"Import")
FldDelimit:=27 ` Set field delimiter to Escape character
RecDelimit:=10 ` Set record delimiter to Line Feed character
IMPORT TEXT([People];"NewPeople.txt") ` Import from "NewPeople.txt" document
```

## System variables and sets

OK is set to 1 if the import is successfully completed; otherwise, it is set to 0.

# Interruptions

- ABORT
- ASSERT
- Asserted
- FILTER EVENT
- Get assert enabled
- GET LAST ERROR STACK
- Method called on error
- Method called on event
- ON ERR CALL
- ON EVENT CALL
- SET ASSERT ENABLED

# ABORT

ABORT
Does not require any parameters

## Description

The **ABORT** command is used from within an error-handling project method installed using the command **ON ERR CALL**.

If you do not have an error-handling project method, when an error occurs (for example, a database engine error) 4D displays its standard error dialog box and then interrupts the execution of your code. If the code being executed is:

- an object method, form method (or a project method called by a form or object method), the control returns to the form currently being displayed.
- a method called from a menu, the control returns to the menu bar or to the form currently being displayed.
- the master method of a process, the process then ends.
- a method called directly or indirectly by an import or export operation, the operation is stopped. The same is true for sequential queries or order by operations.
- And so on...

If you use an error-handling project method to catch errors, 4D neither displays its standard error dialog box nor interrupts the execution of your code. Instead, 4D calls your error-handling project method (that you can see as an exception handler), and resumes the execution to the next line of code in the method that triggered the error.

There are errors you can treat programmatically; for example, during an import operation, if you catch a database engine duplicated value error, you can “cover” the error and pursue the import. However, there are errors that you cannot process and errors that you should not “cover.” In these cases, you need to stop the execution by calling **ABORT** from within the error-handling project method.

## Historical Note

Although the **ABORT** command is intended to be used only from within a error-handling project method, some members of the 4D community also use it to interrupt execution in other project methods. The fact that it works is only a side effect. We do not recommend the use of this command in methods other than error-handling methods.

# ASSERT

ASSERT ( boolExpression {; messageText} )

| Parameter | Type | | Description |
|---|---|---|---|
| boolExpression | Boolean | ➡ | Boolean expression |
| messageText | Text | ➡ | Text of error message |

## Description

The **ASSERT** command evaluates the *boolExpression* assertion passed in parameter and, if it returns false, stops the code execution with an error message. The command works in interpreted and compiled mode.

If *boolExpression* is true, nothing happens. If it is false, the command triggers the error -10518 and displays by default the text of the assertion preceded by the message "Assert failed:". You can intercept this error via a method installed using the **ON ERR CALL** command, in order, for example, to provide info for a log file.

Optionally, you can pass a *messageText* parameter to display a custom error message instead of the text of the assertion.

An assertion is an instruction inserted in the code that is responsible for detecting any anomalies during its execution. The principle consists in verifying that an expression is true at a given moment and, should the opposite occur, to cause an exception. Assertions are above all used to detect cases that should usually not ever occur. They are mainly used to detect programming bugs. It is possible to globally enable or disable all the assertions of an application (for example according to the type of version) via the **SET ASSERT ENABLED** command. For more information about assertions in programming, please refer to the article concerning them on Wikipedia: http://en.wikipedia.org/wiki/Assertion_(computing)

## Example 1

Before carrying out operations on a record, the developer wants to make sure that it is actually loaded in read/write mode:

```
READ WRITE([Table 1])
LOAD RECORD([Table 1])
ASSERT(Not(Locked([Table 1])))
 // triggers error -10518 if record is locked
```

## Example 2

An assertion can allow parameters passed to a project method to be tested in order to detect aberrant values. In this example, a custom warning message is used.

```
 // Method that returns the number of a client according to its name passed in $1
C_TEXT($1) // Name of client
ASSERT($1#"";"Search for a blank client name")
 // A blank name in this case is an aberrant value
 // If the assertion is false, the following will be displayed in the error dialog box:
 // "Assert failed: Search for a blank client name"
```

# Asserted

Asserted ( boolExpression {; messageText} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| boolExpression | Boolean | ⇒ | Boolean expression |
| messageText | Text | ⇒ | Text of error message |
| Function result | Boolean | ⇐ | Result of evaluation of boolExpression |

## Description

The **Asserted** command has an operation similar to that of the **ASSERT** command, with one difference in that it returns a value which is the result of the evaluation of the *boolExpression* parameter. It therefore allows the use of an assertion during the evaluation of a condition (see the example). For more information about the operation of assertions and the parameters of this command, please refer to the description of the **ASSERT** command.

**Asserted** accept a Boolean expression as a parameter and returns the result of the evaluation of this expression. If the expression is false and if the assertions are enabled (see the **SET ASSERT ENABLED** command), the error -10518 is generated, exactly as for the **ASSERT** command. If the assertions are disabled, **Asserted** returns the result of the expression that was passed without triggering an error.

**Note:** Like the **ASSERT** command, **Asserted** works in interpreted and compiled mode.

## Example

Insertion of an assertion in the evaluation of an expression:

```
READ WRITE([Table 1])
LOAD RECORD([Table 1])
If(Asserted(Not(Locked([Table 1]))))
 // This code triggers the error -10518 if the record is locked
   ...
End if
```

# FILTER EVENT

FILTER EVENT
Does not require any parameters

## Description

You call the **FILTER EVENT** command from within an event-handling project method installed using the **ON EVENT CALL** command.

If an event-handling method calls **FILTER EVENT**, the current event is not passed to 4D.

This command removes the current event (i.e., click, keystroke) from the event queue, so 4D will not perform any additional treatment to the one you made in the event-handling project method.

**WARNING:** Do not create an event-handling method that only calls the **FILTER EVENT** command, because all the events are going to be ignored by 4D. When you have an event-handling method with only the **FILTER EVENT** command, type Ctrl+Shift+Backspace (on Windows) or Command-Option-Shift-Control-Backspace (on Macintosh). This converts the On Event Call process into a normal process that does not get any events at all.

**Special case:** The **FILTER EVENT** command can also be used within a standard output form method when the form is displayed using the **DISPLAY SELECTION** or **MODIFY SELECTION** commands. In this specific case, the **FILTER EVENT** command allows you to filter double-clicks on the records (and in this way execute actions other than the opening of records in page mode).

To do this, place the following lines in the output form method:

```
If(Form event=On Double Clicked)
   FILTER EVENT
   ... `Process the double-click
End if
```

## Example

See example for the **ON EVENT CALL** command.

# Get assert enabled

Get assert enabled -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Boolean | True = assertions are enabled False = assertions are disabled |

## Description

The **Get assert enabled** command returns True or False according to whether or not assertions are enabled in the current process. For more information about assertions, please refer to the description of the **ASSERT** command.

By default, assertions are enabled but they may have been disabled using the **SET ASSERT ENABLED** command.

# GET LAST ERROR STACK

GET LAST ERROR STACK ( codesArray ; intCompArray ; textArray )

| Parameter | Type | | Description |
|---|---|---|---|
| codesArray | Longint array | ⇦ | Error numbers |
| intCompArray | String array | ⇦ | Internal component codes |
| textArray | String array | ⇦ | Text of errors |

## Description

The **GET LAST ERROR STACK** command returns information about the current stack of errors of the 4D application. When a 4D statement causes an error, the current error stack contains a description of the error as well as any series of errors generated. For example, a "disk full" type error causes a write error in the file then an error in the record saving command: the stack will therefore contain three errors. If the last 4D statement did not generate an error, the current error stack is empty.

This generic command can be used to process any type of error that may occur in the 4D application.

**Note:** However, to obtain detailed information concerning the errors generated by an ODBC source, it will be necessary to use the **SQL GET LAST ERROR** command.

This command must be called from an on error call method installed by the **ON ERR CALL** command.

The information is returned in three synchronized arrays:

- *codesArray*: This array receives the list of error codes generated.
- *intCompArray*: This array contains the codes of the internal components associated with each error.
- *textArray*: This array contains the text of each error.

The list of error codes and their text is provided in the sections of the **Error Codes** theme.

# Method called on error

Method called on error -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String | ⮌ | Name of method called on error |

## Description

The **Method called on error** command returns the name of the method installed by the **ON ERR CALL** command for the current process.

If no such method has been installed, an empty string ("") is returned.

## Example

This command is particularly useful in the context of components because it enables you to temporarily change and then restore the error-catching methods:

```
$methCurrent:=Method called on error
ON ERR CALL("NewMethod")
 ` If the document cannot be opened, an error is generated
$ref:=Open document("MyDocument")
 ` Reinstallation of previous method
ON ERR CALL($methCurrent)
```

# Method called on event

Method called on event -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String | | Name of method called on event |

## Description

The **Method called on event** command returns the name of the method installed by the **ON EVENT CALL** command.
If no such method has been installed, an empty string ("") is returned.

# ON ERR CALL

ON ERR CALL ( errorMethod )

| Parameter | Type | | Description |
|---|---|---|---|
| errorMethod | String | ➡ | Error method to be invoked, or Empty string to stop trapping errors |

## Description

The **ON ERR CALL** command installs the project method, whose name you pass in *errorMethod*, as the method for catching (trapping) errors. This project method is called the **error-handling method** or **error-catching method**.

Once an error-handling project is installed, 4D calls the method each time an error occurs during the execution of a 4D language command.

The scope of this command is the current process. You can have only one error-handling method per process at a time, but you can have different error-handling methods for several processes.

**Notes:**

- In the context of the use of components, this command applies to the current database only. If an error is generated from a component, *errorMethod* is not called in the host database.
- If **ON ERR CALL** is called from a process for which you requested preemptive execution (in 64-bit compiled mode), the compiler checks whether *errorMethod* is thread-safe and returns errors if it is not compatible with the preemptive mode. For more information, refer to the **Preemptive 4D processes** section.

To stop the trapping of errors, call **ON ERR CALL** again and pass the empty string in *errorMethod*.

You can identify errors by reading the *Error* system variable, which contains the code number of the error. Error codes are listed in the **Error Codes** theme. For example, you can see the section **Syntax Errors (1 -> 81)**. The Error variable value is significant only within the error-handling method; if you need the error code within the method that provoked the error, copy the *Error* variable to your own process variable. You can also access the *Error method*, *Error line* and *Error formula* system variables which contain, respectively, the name of the method, the line number and the text of the formula where the error occurred (see **Error, Error method, Error line**).

You can use the **GET LAST ERROR STACK** command to obtain the error sequence (i.e., the error "stack") at the origin of the interruption.

The error-handling method should manage the error in an appropriate way or present an error message to the user. Errors can be generated during processing performed by:

- The 4D database engine; for example, when saving a record causes the violation of a trigger rule.
- The 4D environment; for example, when you do not have enough memory for allocating an array.
- The operating system on which the database is run; for example, disk full or I/O errors.

The **ABORT** command can be used to terminate processing. If you don't call **ABORT** in the error-handling method, 4D returns to the interrupted method and continues to execute the method. Use the **ABORT** command when an error cannot be recovered.

If an error occurs in the error-handling method itself, 4D takes over error handling. Therefore, you should make sure that the error-handling method cannot generate an error. Also, you cannot use **ON ERR CALL** inside the error-handling method.

## Example 1

The following project method tries to create a document whose name is received as parameter. If the document cannot be created, the project metod returns 0 (zero) or the error code:

```
 //Create doc project method
 //Create doc ( String ; Pointer ) -> LongInt
 //Create doc ( DocName ; ->DocRef ) -> Error code result

gError:=0
ON ERR CALL("IO ERROR HANDLER")
$2->:=Create document($1)
```

```
ON ERR CALL("")
$0:=gError
```

The ***IO ERROR HANDLER*** project method is listed here:

```
 //IO ERROR HANDLER project method
gError:=Error //just copy the error code to the process variable gError
```

Note the use of the *gError* process variable to get the error code result within the current executing method. Once these methods are present in your database, you can write:

```
 // ...
C_TIME(vhDocRef)
$vlErrCode:=Create doc($vsDocumentName;->vhDocRef)
If($vlErrCode=0)
 //...
   CLOSE DOCUMENT($vlErrCode)
Else
   ALERT("The document could not be created, I/O error "+String($vlErrCode))
End if
```

## Example 2

See example in the **Arrays and Memory** section.

## Example 3

While implementing a complex set of operations, you may end up with various subroutines that require different error-handling methods. You can have only one error-handling method per process at a time, so you have two choices:
- Keep track of the current one each time you call **ON ERR CALL**, or
- Use a process array variable (in this case, *asErrorMethod*) to "pile up" the error-handling methods and a project method (in this case, ***ON ERROR CALL***) to install and deinstall the error-handling methods.

You must initialize the array at the very beginning of the process execution:

```
 ` Do NOT forget to initialize the array at the beginning
 ` of the process method (the project method that runs the process)
ARRAY STRING(63;asErrorMethod;0)
```

Here is the custom ***ON ERROR CALL*** method:

```
 ` ON ERROR CALL project method
 ` ON ERROR CALL { ( String ) }
 ` ON ERROR CALL { ( Method Name ) }

C_STRING(63;$1;$ErrorMethod)
C_LONGINT($vlElem)

If(Count parameters>0)
   $ErrorMethod:=$1
Else
   $ErrorMethod:=""
End if

If($ErrorMethod#"")
   C_LONGINT(gError)
   gError:=0
   $vlElem:=1+Size of array(asErrorMethod)
   INSERT IN ARRAY(asErrorMethod;$vlElem)
   asErrorMethod{$vlElem}:=$1
   ON ERR CALL($1)
Else
   ON ERR CALL("")
   $vlElem:=Size of array(asErrorMethod)
   If($vlElem>0)
      DELETE FROM ARRAY(asErrorMethod;$vlElem)
```

```
      If($vlElem>1)
         ON ERR CALL(asErrorMethod{$vlElem-1})
      End if
   End if
End if
```

Then, you can call it this way:

```
gError:=0
ON ERROR CALL("IO ERRORS") ` Installs the IO ERRORS error-handling method
 ` ...
ON ERROR CALL("ALL ERRORS") ` Installs the ALL ERRORS error-handling method
 ` ...
ON ERROR CALL ` Deinstalls ALL ERRORS error-handling method and reinstalls IO ERRORS
 ` ...
ON ERROR CALL ` Deinstalls the IO ERRORS error-handling method
 ` ...
```

## Example 4

The following error-handling method ignores the user interruptions and displays the error text:

```
 //Show_errors_only project method
If(Error#1006) //this is not a user interruption
   ALERT("The error "+String(Error)+" occurred. The code in question is: ¥""+Error formula+"¥"")
End if
```

# ON EVENT CALL

ON EVENT CALL ( eventMethod {; processName} )

| Parameter | Type | | Description |
|---|---|---|---|
| eventMethod | String | ⇒ | Event method to be invoked, or Empty string to stop intercepting events |
| processName | String | ⇒ | Process name |

## Description

The **ON EVENT CALL** command installs the method, whose name you pass in *eventMethod*, as the method for catching (trapping) events. This method is called the **event-handling method** or **event-catching method**.

**Tip:** This command requires advanced programming knowledge. Usually, you do not need to use **ON EVENT CALL** for working with events. While using forms, 4D handles the events and sends them to the appropriate forms and objects.

**Tip:** Commands such as **GET MOUSE**, **Shift down,** etc., can be used for getting information about events. These commands can be called from within object methods to get the information you need about an event involving an object. Using them spares you the writing of an algorithm based on the **ON EVENT CALL** scheme.

The scope of this command is the current working session. By default, the method is run in a separate local process. You can have only one event-handling method at a time. To stop catching events with a method, call **ON EVENT CALL** again and pass an empty string in *eventMethod*.

Since the event-handling method is run in a separate process, it is constantly active, even if no 4D method is running. After installation, 4D calls the event-handling method each time an event occurs. An event can be a mouse click or a keystroke.

The optional *processName* parameter names the process created by the **ON EVENT CALL** command. If *processName* is prefixed with a dollar sign (*$*), a local process is started, which is usually what you want. If you omit the *processName* parameter, 4D creates, by default, a local process named *$Event Manager*.

**WARNING:** Be very careful in what you do within an event-handling method. Do NOT call commands that generate events, otherwise it will be extremely difficult to get out of the event-handling method execution. The key combination **Ctrl+Shift+Backspace** (on Windows) or **Command-Shift-Control-Backspace** (on Macintosh) allows you to kill the Event Manager process. You may want to use this technique to recover an event-handling gone wrong (i.e., one that has bugs triggering events).

In the event-handling method, you can read the following system variables—*MouseDown, KeyCode, Modifiers, MouseX, MouseY* and *MouseProc*. Note that these variables are process variables. Their scope is therefore the event-handling process. Copy them into interprocess variables if you want their values available in another process.

- The *MouseDown* system variable is set to 1 if the event is a mouse click, and to 0 if it is not.
- The *KeyCode* system variable is set to the code for a keystroke. This variable may return an character code or a function key code. These codes are listed in the sections **Unicode Codes** and **EXPORT TEXT** (and its subsections) as well as in the section **Function Key Codes**. 4D provides predefined constants for the major ASCII Codes and for Function Keys. In the Explorer window, look for the themes of these constants.
- The *Modifiers* system variable contains the modifier value. It indicates whether a modifier key was down when the event occurred. The following keys can be detected:

| Platform | Modifiers |
|---|---|
| Windows | Shift key, Caps Lock, Alt key, Ctrl key |
| Macintosh | Shift key, Caps Lock, Alt (or Option) key, Command key, Ctrl key |

The modifier keys do not generate an event on their own; another key or the mouse button must also be pressed. The **Modifiers** variable is a Long Integer variable containing a bit field. 4D provides predefined constants specifying the bit position or bit mask for each modifier key. For example, to detect if the Shift key was pressed for the event, you can write either:

```
If(Modifiers?? Shift key bit) //If the Shift key was down
```

or:

```
If((Modifiers & Shift key mask)#0) //If the Shift key was down
```

You can use one of the following constants, depending on the modifier key to be tested and the platform, which are found in the **Events (Modifiers)** theme:

| Modifier | Constant |
|---|---|
| Shift | Shift key bit / Shift key mask |
| Caps Lock | Caps lock key bit / Caps lock key mask |
| Alt (also called Option under OS X) | Option key bit / Option key mask |
| Ctrl under Windows | Command key bit / Command key mask |
| Ctrl under OS X | Control key bit / Control key mask |
| Command under OS X | Command key bit / Command key mask |
| Right click | Control key bit / Control key mask |

- The system variables *MouseX* and *MouseY* contain the horizontal and vertical positions of the mouse click, expressed in the local coordinate system of the window where the click occurred. The upper left corner of the window is position 0,0. These are meaningful only when there is a mouse click.
- The **MouseProc** system variable contains the process reference number of the process in which the event occurred (mouse click).

**Important:** The system variables **MouseDown**, **KeyCode**, **Modifiers**, **MouseX**, **MouseY**, and **MouseProc** contain significant values only within an event-handling method installed with **ON EVENT CALL**.

## Example

This example will cancel printing if the user presses **Ctrl+period**. First, the event-handling method is installed. Then a message is displayed, announcing that the user can cancel printing. If the interprocess variable *◊vbWeStop* is set to True in the event-handling method, the user is alerted to the number of records that have already been printed. Then the event-handling method is deinstalled:

```
PAGE SETUP
If(OK=1)
   ◊vbWeStop:=False
   ON EVENT CALL("EVENT HANDLER") ` Installs the event-handling method
   ALL RECORDS([People])
   MESSAGE("To interrupt printing press Ctrl+Period")
   $vlNbRecords:=Records in selection([People])
   For($vlRecord;1;$vlNbRecords)
      If(◊vbWeStop)
         ALERT("Printing cancelled at record "+String($vlRecord)+" of "+String($vlNbRecords))
         $vlRecord:=$vlNbRecords+1
      Else
         Print form([People];"Special Report")
      End if
   End for
   PAGE BREAK
   ON EVENT CALL("") ` Deinstalls the event-handling method
End if
```

If **Ctrl+period** has been pressed, the event-handling method sets *◊vbWeStop* to True:

```
 ` EVENT HANDLER project method
If((Modifiers?? Command key bit) & (KeyCode=Period))
   CONFIRM("Are you sure?")
   If(OK=1)
      ◊vbWeStop:=True
      FILTER EVENT ` Do NOT forget this call; otherwise 4D will also get this event
   End if
End if
```

Note that this example uses **ON EVENT CALL** because it performs a special printing report using the **PAGE SETUP**, **Print form** and **PAGE BREAK** commands with a **For...End for** loop.

If you print a report using **PRINT SELECTION**, you do NOT need to handle events that let the user interrupt the printing; this command does that for you.

# SET ASSERT ENABLED

SET ASSERT ENABLED ( assertions {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| assertions | Boolean | ⇒ | True = enable assertions False = disable assertions |
| * | Operator | ⇒ | If omitted = command applies to all the processes (existing or created subsequently) If passed= command applies to current process only |

## Description

The **SET ASSERT ENABLED** command can be used to disable or re-enable any assertions inserted into the 4D code of the application. For more information about assertions, please refer to the description of the **ASSERT** command.

By default, assertions added in the program are enabled in interpreted and compiled mode. This command is useful when you want to disable them since their evaluation can sometimes be costly in terms of execution time and you may also want them to be hidden from the final user of the application. Typically, the **SET ASSERT ENABLED** command could be used in the **On Startup database method** in order to enable or disable assertions according to whether the application is in "Test" mode or in "Production" mode.

By default, the **SET ASSERT ENABLED** command affects all the processes of the application. To restrict the effect of the command to the current process only, you can pass the * parameter.

Please note that when assertions are disabled, expressions passed to **ASSERT** commands are no longer evaluated. The lines of code that call this command no longer have any effect on the operation of the application, neither in terms of behavior, nor in terms of performance.

## Example

Disabling assertions:

```
SET ASSERT ENABLED(False)
ASSERT(TestMethod) // TestMethod will not be called since assertions are disabled
```

# JSON

- Overview of JSON commands
- JSON Parse
- JSON PARSE ARRAY
- JSON Stringify
- JSON Stringify array
- JSON TO SELECTION
- Selection to JSON

# Overview of JSON commands

JSON commands generate and parse JSON-format language objects. More particularly, JSON format makes it possible to access 4D databases (data and structure) using a Web browser.

Support for structured objects is a major new feature of the language in 4D v14, intended to facilitate the exchange of structured data. Thanks to the commands of the "JSON" theme, 4D can work directly with JSON objects. However, 4D can also work with "native" objects (whose structure is inspired by JSON), allowing exchanges with all types of language. For more information, refer to the **Objects (Language)** chapter.

## Introduction to JSON

"JSON or JavaScript Object Notation is a generic text-based data format derived from object notation of the ECMAScript language." (*source: Wikipedia*). JSON is independent from any other language, but uses conventions that are familiar to programmers using C++ or JavaScript, Perl, and so on. It is a format that is particularly suitable for data exchange.

This section summarizes the notation principles implemented in JSON. For a complete description of this format, refer the following site: www.json.org/index.html.

### JSON syntax

JSON syntax is based on the following principles:

- data consists of name/value pairs,
- data is separated by commas,
- objects are defined by braces {},
- arrays are defined by brackets [ ].

### JSON properties

JSON data is expressed in name/value (or key/value) pairs. A name/value pair contains a field name (in quotes), then a colon, followed by a value. For example:

```
"firstName":"John"
```

For information, this example is equivalent to the following in JavaScript:

```
firstName="John"
```

Keep in mind that property names are diacritical and case-sensitive. If you write "FirstName" instead of "firstName," this gives you a new name/value pair.

### JSON data types

The following types of values are supported in JSON:

| Type | Description | Comments |
| --- | --- | --- |
| string | Any Unicode character except for " and ¥<br>Values, like property names, are in quotes ("), for example, "city":"Paris" | ¥ is used for control characters:<br>¥" = quotes<br>¥¥ = backslash<br>¥/ = slash<br>¥b = backspace<br>¥f = formfeed<br>¥n = line break<br>¥r = carriage return<br>¥t = tab<br>¥u = four hexadecimal digits |
| number | Integer or floating point number | Number similar to a C or Java number, except that the octal and hexadecimal formats are not used |
| object | { } | |
| array | [ ] | |
| boolean | true or false | |
| null | null | |

### JSON objects

JSON objects are defined by braces and can contain an undefined number of name/value pairs, for example:

```
{ "firstName":"John" , "lastName":"Doe" }
```

### JSON arrays

JSON arrays are defined by brackets. Each array can contain an undefined number of objects:

```
{    "employees": [         { "firstName":"John" , "lastName":"Doe" },         { "firstName":"Anna" , "lastName":"Smith" },         { "firstName":"Peter" , "lastName":"Jones" }     ] }
```

## Time zone support

By default, when 4D dates are converted to and from JSON, they take into account the time zone of the machine where the conversion took place (in conformity with JavaScript). For example, in France (GMT+2), converting !23/08/2013! gives you "2013-08-22T22:00:00Z" and vice versa.

You can change this functioning and no longer take the time zone into account, during the implementation of export procedures for example, using the **SET DATABASE PARAMETER** command.

For more information about converting 4D/JSON dates, refer to **Conversion of JavaScript dates**.

# JSON Parse

JSON Parse ( jsonString {; type} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| jsonString | String | ➡ | JSON string to parse |
| type | Longint | ➡ | Type in which to convert the values |
| Function result | Boolean, Object, Pointer, Real, Text | ⬅ | Values extracted from JSON string |

## Description

The **JSON Parse** command parses the contents of a JSON-formatted string and extracts values that you can store in a 4D field or variable. This command deserializes JSON data; it performs the opposite action of the **JSON Stringify** command.

In *jsonString*, pass the JSON-formatted string whose contents you want to parse. This string must be formatted correctly, otherwise a parsing error is generated. **JSON Parse** can therefore be used to validate JSON strings.

**Note:** If you use pointers, you must call the **JSON Stringify** command before calling **JSON Parse**.

By default, if you omit the *type* parameter, 4D attempts to convert the value obtained into the type of the variable or field used to store the results (if one is defined). Otherwise, 4D attempts to infer its type. You can also force the type interpretation by passing the *type* parameter: pass one of the following constants, available in the **Field and Variable Types** theme:

| Constant | Type | Value |
|---|---|---|
| Is Boolean | Longint | 6 |
| Is date | Longint | 4 |
| Is longint | Longint | 9 |
| Is object | Longint | 38 |
| Is real | Longint | 1 |
| Is text | Longint | 2 |

**Notes:**

- Real type values must be included in the range ±10.421e±10
- In text type values, all special characters must be escaped, including quotes (see examples)
- JSON dates must be in the format "¥"YYYY-MM-DDTHH:mm:ssZ¥"". The command considers that the 4D date contains a local time and not GMT.

## Example 1

Examples of simple conversions:

```
C_REAL($r)
$r:=JSON Parse("42.17") //$r = 42,17 (Real)

C_LONGINT($el)
$el:=JSON Parse("120.13";Is longint) //$el=120

C_TEXT($t)
$t:=JSON Parse("¥"Year 42¥"";Is text) // $t="Year 42" (text)

C_OBJECT($o)
$o:=JSON Parse("{¥"name¥":¥"jean¥"}")
 // $o = {"name":"john"} (4D object)

C_BOOLEAN($b)
$b:=JSON Parse("{¥"manager¥":true}";Is Boolean) // $b=true
```

## Example 2

Example of converting date type data:

```
$test:=JSON Parse("¥"1990-12-25T12:00:00Z¥"")
 // $test=1990-12-25T12:00:00Z
C_DATE($date)
$date:=JSON Parse("¥"2008-01-01T12:00:00Z¥"";Is date)
 //$date=01/01/08
```

## Example 3

This example shows the combined use of the **JSON Stringify** and **JSON Parse** commands:

```
C_TEXT($MyContact)
C_OBJECT($Contact)

 // JSON Stringify: conversion of JSON object into a JSON string
$MyContact:=JSON Stringify("{¥"name¥":¥"Monroe¥",¥"firstname¥":¥"Alan¥"}")
 // $MyContact = "{¥¥"name¥¥":¥¥"Monroe¥¥",¥¥"firstname¥¥":¥¥"Alan¥¥"}"
 // JSON Parse: conversion of JSON string into a JSON object
$Contact:=JSON Parse("{¥"name¥":¥"Monroe¥",¥"firstname¥":¥"Alan¥"}")
 // $Contact = {"name":"Monroe","firstname":"Alan"}
```

# JSON PARSE ARRAY

JSON PARSE ARRAY ( jsonString ; objArray )

| Parameter | Type | | Description |
|---|---|---|---|
| jsonString | String | ➡ | JSON string to parse |
| objArray | Object array, Text array, Real array, Boolean array, Pointer array | ⬅ | Array containing result from parsing of JSON string |

## Description

The **JSON PARSE ARRAY** command parses the contents of a JSON-formatted string and puts the data extracted into the *objArray* array. This command deserializes the JSON data; it performs the opposite action of the **JSON Stringify array** array command.

In *jsonString*, pass the JSON-formatted string whose contents you want to parse. This string must be formatted correctly, otherwise a parsing error is generated.

In *objArray*, pass the object to receive the parsing results.

## Example

In this example, data from fields of the records in a table are extracted and then placed in object arrays:

```
C_OBJECT($ref)
ARRAY OBJECT($sel;0)
ARRAY OBJECT($sel2;0)
C_TEXT(v_String)

OB SET($ref;"name";->[Company]Company Name)
OB SET($ref;"city";->[Company]City)

While(Not(End selection([Company])))
   $ref_company:=OB Copy($ref;True)
   APPEND TO ARRAY($sel;$ref_company)
 // $sel{1}={"name":"4D SAS","city":"Clichy"}
 // $sel{2}={"name":"MyComp","city":"Lyon"}
 // ...
   NEXT RECORD([Company])
End while

v_String:=JSON Stringify array($sel)
 // v_String= [{"name":"4D SAS","city":"Clichy"},{"name":"MyComp","city":"Lyon"}...]
JSON PARSE ARRAY(v_String;$sel2)
 // $sel2{1}={"name":"4D SAS","city":"Clichy"}
 // $sel2{2}={"name":"MyComp","city":"Lyon"}
 //...
```

# JSON Stringify

JSON Stringify ( value {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| value | Object, Object array, String, Real, Date, Time | ➡ | Data to convert into JSON string |
| * | Operator | ➡ | Pretty printing |
| Function result | Text | ⮌ | String containing serialized JSON text |

## Description

The **JSON Stringify** command converts the *value* parameter into a JSON string. This command serializes data into JSON; it performs the opposite action of the **JSON Parse** command.

Pass the data to be serialized in *value*. It can be expressed in scalar form (string, number, date or time) or by means of a 4D object (or an object array).

In the case of an object, you can include all types of values (see the **JSON data types** paragraph). JSON formatting must respect the following rules:

- String values must be enclosed in quotes. All Unicode character can be used except for special characters that must be preceded by a backslash.
- Numbers: interval of ±10.421e±10
- Booleans: "true" or "false" strings
- Pointers to a field, variable or array (the pointer is evaluated when it is stringified)
- Dates: Text type
- Times: Real type

You can pass the optional * parameter to include formatting characters in the resulting string. This improves the presentation of JSON data (known as pretty formatting).

## Example 1

Conversion of scalar values:

```
$vc:=JSON Stringify("Eureka!") // "Eureka!"
$vel:=JSON Stringify(120) // "120"
$vd:=JSON Stringify(!28/08/2013!) //  "2013-08-27T22:00:00Z"
$vh:=JSON Stringify(?20:00:00?) // "72000000" seconds since midnight
```

## Example 2

Conversion of a string containing special characters:

```
$s:=JSON Stringify("{¥"name¥":¥"john¥"}")
 // $s="{¥¥"name¥¥":¥¥"john¥¥"}"
$p:=JSON Parse($s)
 // $p={"name":"john"}
```

## Example 3

Example using a pointer to a variable:

```
C_OBJECT($MyTestVar)
C_TEXT($name ;$jsonstring )
OB SET($MyTestVar;"name";->$name) // object definition
 // $MyTestVar= {"name":"->$name"}

$jsonstring :=JSON Stringify($MyTestVar)
```

```
 // $jsonstring ="{"name":""}"
 //...

$name:="Smith"
$jsonstring :=JSON Stringify($MyTestVar)
 //$jsonstring = "{"name" : "Smith"}"
```

## Example 4

Serialization of a 4D object:

```
C_TEXT($varjsonTextserialized)
C_OBJECT($Contact)
OB SET($Contact;"firstname";"Alan")
OB SET($Contact;"lastname";"Monroe")
OB SET($Contact;"age";40)
OB SET($Contact;"phone";"[555-0100,555-0120]")

$varjsonTextserialized:=JSON Stringify($Contact)

 // $varjsonTextserialized = "{"lastname":"Monroe","phone":"[555-0100,
 // 555-0120]","age":40,"firstname":"Alan"}"
```

## Example 5

Examples of serializing a 4D object with and without the * parameter:

```
C_TEXT($MyContact)
C_TEXT($MyPContact)
C_OBJECT($Contact;$Children)
OB SET($Contact;"lastname";"Monroe";"firstname";"Alan")
OB SET($Children;"firstname";"Jim";"age";"12")
OB SET($Contact;"children";$Children)
$MyContact:=JSON Stringify($Contact)
$MyPContact:=JSON Stringify($Contact;*)
 //$MyContact= {"lastname":"Monroe","firstname":"Alan","children":{"firstname":"John","age":"12"}}
 //$MyPContact= {¥n¥t"lastname": "Monroe",¥n¥t"firstname": "Alan",¥n¥t"children": {¥n¥t¥t"firstname": "John",¥n¥t¥t"age":
"12"¥n¥t}¥n}
```

The advantage of this formatting is clear when the JSON is shown in a Web area:

- Standard formatting:

```
{"Name":"Monroe","firstname":"Alan","children":{"firstname":"John","age":12}}
```

- Pretty formatting:

```
{
	"Name": "Monroe",
	"firstname": "Alan",
	"children": {
		"firstname": "John",
		"age": 12
	}
}
```

# JSON Stringify array

JSON Stringify array ( array {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| array | Text array, Real array, Boolean array, Pointer array, Object array | ➡ | Array whose contents must be serialized |
| * | Operator | ➡ | Pretty formatting |
| Function result | Text | ⮌ | String containing the serialized JSON array |

## Description

The **JSON Stringify array** command converts the 4D array *array* into a serialized JSON array. . This command performs the opposite action of the **JSON PARSE ARRAY** command.

In *array*, pass a 4D array containing the data to be serialized. This array may be of the text, real, Boolean, pointer or object type.

You can pass the optional * parameter to use pretty formatting in the resulting string. This improves the presentation of JSON data by including formatting characters when it is displayed in a Web page.

## Example 1

Conversion of a text array:

```
C_TEXT($jsonString)
ARRAY TEXT($ArrayFirstname;2)
$ArrayFirstname{1}:="John"
$ArrayFirstname{2}:="Jim"
$jsonString :=JSON Stringify array($ArrayFirstname)

 // $jsonString = "["John","Jim"]"
```

## Example 2

Conversion of a text array containing numbers:

```
ARRAY TEXT($phoneNumbers;0)
APPEND TO ARRAY($phoneNumbers ;"555-0100")
APPEND TO ARRAY($phoneNumbers ;"555-0120")
$string :=JSON Stringify array($phoneNumbers)
 // $string = "["555-0100","555-0120"]"
```

## Example 3

Conversion of an object array:

```
C_OBJECT($ref_john)
C_OBJECT($ref_jim)
ARRAY OBJECT($myArray;0)
OB SET($ref_john;"name";"John";"age";35)
OB SET($ref_jim;"name";"Jim";"age";40)
APPEND TO ARRAY($myArray ;$ref_john)
APPEND TO ARRAY($myArray ;$ref_jim)
$JsonString :=JSON Stringify array($myArray)
 // $JsonString = "[{"name":"John","age":35},{"name":"Jim","age":40}]"

 // If you want to view the result in a Web page,
 // pass the optional * parameter:
$JsonStringPretty :=JSON Stringify array($myArray;*)
```

```
[
  {
    "name": "John",
    "age": 35
  },
  {
    "name": "Jim",
    "age": 40
  }
]
```

## Example 4

Conversion of a 4D selection in an object array:

```
C_OBJECT($jsonObject)
C_TEXT($jsonString)

QUERY([Company];[Company]Company Name="a@")
OB SET($jsonObject;"company name";->[Company]Company Name)
OB SET($jsonObject;"city";->[Company]City)
OB SET($jsonObject;"date";[Company]Date_input)
OB SET($jsonObject;"time";[Company]Time_input)
ARRAY OBJECT($arraySel;0)

While(Not(End selection([Company])))
   $ref_value:=OB Copy($jsonObject;True)
 // If you do not copy them, the values will be empty strings
   APPEND TO ARRAY($arraySel;$ref_value)
 // Each element contains the selected values, for example:
 // $arraySel{1} = // {"company name":"APPLE","time":43200000,"city":
 // "Paris","date":"2012-08-02T00:00:00Z"}
   NEXT RECORD([Company])
End while

$jsonString:=JSON Stringify array($arraySel)
 // $jsonString = "[{"company name":"APPLE","time":43200000,"city":
 //"Paris","date":"2012-08-02T00:00:00Z"},{"company name":
 //"ALMANZA",...}]"
```

# JSON TO SELECTION

JSON TO SELECTION ( aTable ; jsonArray )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | 4D table into which elements are copied |
| jsonArray | Text | ⇒ | Array of objects in JSON |

## Description

The **JSON TO SELECTION** command copies the contents of an array of JSON objects *jsonArray* to the selection of records of *aTable*.

The *jsonArray* parameter is a *text* representing an array of objects formatted in JSON and containing one or more elements. The expected syntax is of the type:

```
"[{"attribute1":"value1","attribute2":"value2",...},...,{"attribute1":"valueN","attribute2":"valueN",...}]"
```

If a selection exists for *aTable* at the time of the call, the elements of the JSON array are copied into the records based on the order of the array and the order of the records. If the number of elements defined in the JSON array is greater than the number of records in the current selection, new records are created. The records, whether they are new or existing, are automatically saved.

**Note:** This command supports Object type fields: JSON data is converted automatically.

**Warning:** Since **JSON TO SELECTION** replaces any information found in the existing records, this command must be used with caution.

If a record is locked by another process during the execution of the command, it is not modified. All the locked records are placed in the **The LockedSet System Set**. After the execution of **JSON TO SELECTION**, you can test whether the *LockedSet* set contains any records that were locked.

## Example

Using the **JSON TO SELECTION** command to add records to the [Company] table:

```
C_OBJECT($Object1;$Object2;$Object3;$Object4)
C_TEXT($ObjectString)
ARRAY OBJECT($arrayObject;0)

OB SET($Object1;"ID";"200";"Company Name";"4D SAS";"City";"Clichy")
APPEND TO ARRAY($arrayObject;$Object1)

OB SET($Object2;"ID";"201";"Company Name";"APPLE";"City";"Paris")
APPEND TO ARRAY($arrayObject;$Object2)

OB SET($Object3;"ID";"202";"Company Name";"IBM";"City";"London")
APPEND TO ARRAY($arrayObject;$Object3)

OB SET($Object4;"ID";"203";"Company Name";"MICROSOFT";"City";"New York")
APPEND TO ARRAY($arrayObject;$Object4)

$ObjectString:=JSON Stringify array($arrayObject)

 // $ObjectString = "[{"ID":"200","City":"Clichy","Company Name":"4D
 // SAS"},{"ID":"201","City":"Paris","Company Name":"APPLE"},{"ID":"202",
 //"City":"London","Company Name":"IBM"},{"ID":"203","City":"New
 //York","Company Name":"MICROSOFT"}]"

JSON TO SELECTION([Company];$ObjectString)
 // You create 4 records in the [Company] table, filling the ID,
 //Company name and city fields
```

# Selection to JSON

Selection to JSON ( aTable {; aField}{; aField2 ; ... ; aFieldN}{; template}) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to serialize |
| aField | Field | ⇒ | Field(s) whose contents must be serialized |
| template | Object | ⇒ | Object for selection of labels and fields |
| Function result | Text | ⇐ | String containing serialized JSON array |

## Description

The **Selection to JSON** command returns a string containing a JSON array with as many elements as there are records in the current selection of *aTable*. Each element of the array is a JSON object containing the labels and values of the fields of the selection.

If you only pass the *aTable* parameter, the command includes, in the JSON array, the values of all the fields of the table that can be expressed in JSON. BLOB and Picture type fields are ignored.

If you do not want to include all the fields of *aTable*, you can use either the *aField* parameter or the *template* parameter:

- *aField*: pass one or more fields in this parameter. Only the values of the fields defined are included in the JSON array.
- *template*: pass a 4D object containing one or more *name/value* pairs where the *name* can be any valid attribute name and the *value* contains a pointer to a field you want to include. This syntax allows you to customize the labels of fields in the JSON array.

This command supports Object type fields: the data of these fields is automatically converted to the JSON format. Note that the following 4D statement will be interpreted as "produce JSON from all values of *objectField* in the current selection of the table":

```
Selection to JSON([aTable];objectField)
```

## Example 1

You want to create a JSON string representing this selection:

HierList - Members: 6 of 6

| Last name : | First name : | Address : | City : | Zip Code : |
|---|---|---|---|---|
| Durant | Mark | 25 Park St | Pittsburgh | 15205 |
| Smith | John | 24 Philadelphia Ave | Dallas | 75203 |
| Anderson | Adeline | 37 Market St | Cincinnati | 45205 |
| Peterson | Paul | 32 South Main St | Dallas | 75203 |
| Harper | Harry | 233 Southport Ave | Cincinnati | 45206 |
| Trace | Sandra | 332 Court St | Pittsburgh | 15205 |

1) You want to include the values of all the fields of the [Members] table:

```
$jsonString :=Selection to JSON([Members])
 // $jsonString =[{"LastName":"Durant","FirstName":"Mark","Address":
 //"25 Park St","Zip code":"15205","City":"Pittsburgh"},{"LastName":
 //"Smith","FirstName":"John","Address":"24 Philadelphia Ave","Zip code":
 //"75203","City":"Dallas"},{"LastName":"Anderson","FirstName"
 //:"Adeline","Address":"37 Market St","Zip code":"45205","City":"Cincinnati"},...]
```

2) You want to reduce the selection and only include two fields in the JSON string by using the syntax based on fields:

```
QUERY([Members];[Members]LastName="A@")
$jsonString :=Selection to JSON([Members];[Members]LastName;[Members]City)
 // $jsonString = [{"LastName":"Anderson","City":"Cincinnati"},{"LastName":"Albert","City":"Houston"}]
```

3) You only want to include one field in the JSON string and use a different label.
You can use the *template* syntax:

```
C_OBJECT($template)
OB SET($template;"Member";->[Members]LastName) //custom label and a single field
```

```
ALL RECORDS([Members])
$jsonString :=Selection to JSON([Members];$template)
 // $jsonString = [{"Member":"Durant"},{"Member":"Smith"},{"Member":"Anderson"},
 // {"Member":"Albert"},{"Member":"Leonard"},{"Member":"Pradel"}]
```

## Example 2

You can use the *template* syntax in order to export fields from different tables:

```
C_OBJECT($template)
C_TEXT($jsonString)
OB SET($template;"Last name";->[Emp]LastName)
OB SET($template;"First name";->[Emp]FirstName)
OB SET($template;"Company";->[Company]LastName) //custom label otherwise conflict with [Emp]LastName field
ALL RECORDS([Emp])
SET FIELD RELATION([Emp]UUID_Company;Automatic;Do not modify)
$jsonString:=Selection to JSON([Emp];$template)
SET FIELD RELATION([Emp]UUID_Company;Structure configuration;Do not modify)
```

# Language

- Command name
- Count parameters
- Current method name
- EXECUTE METHOD
- EXECUTE METHOD IN SUBFORM
- Get pointer
- Is a variable
- Nil
- NO TRACE
- RESOLVE POINTER
- Self
- TRACE
- Type

# Command name

Command name ( command {; info {; theme}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| command | Longint | ⇒ | Command number |
| info | Longint | ⇐ | Thread-safety property of command |
| theme | Text | ⇐ | Language theme of command |
| Function result | String | ⇒ | Localized command name |

## Description

The **Command name** command returns the name as well as (optionally) the properties of the command whose command number you pass in *command*.

**Note:** The number of each command is indicated in the Explorer as well as in the Properties area of this documentation.

**Compatibility note:** Since a command name may vary from one 4D version to the next (commands renamed), this command was used in previous versions of the program to designate a command directly by means of its number, especially in non-tokenized portions of code. This need has diminished over time as 4D continues to evolve because, for non-tokenized statements (formulas), 4D now provides a token syntax. This syntax allows you to avoid potential problems due to variations in command names as well as other elements such as tables, while still being able to type these names in a legible manner (for more information about this point, refer to the **Using tokens in formulas** section). Furthermore, by default, the English language version is used starting with 4D v15; however, the "Use regional system settings" option on the **Methods Page** of the Preferences allows you to continue using the French language in a French version of 4D.

Two optional parameters are available:

- *info*: properties of the command. The returned value is a *bit field*, where currently only the first bit (bit 0) is meaningful. It is set to 1 if the command is thread-safe (i.e., compatible with execution in a preemptive process) and 0 if it is thread-unsafe. Only thread-safe commands can be used in preemptive processes. For more information about this point, refer to the [#title id="8733"/] section.
- *theme*: returns the name of the 4D language theme for the command.

The **Command name** command sets the *OK* variable to 1 if *command* corresponds to an existing command number, and to 0 otherwise. Note, however, that some existing commands have been disabled, in which case **Command name** returns an empty string (see last example).

## Example 1

The following code allows you to load all valid 4D commands in an array:

```
C_LONGINT($Lon_id)
C_TEXT($Txt_command)
ARRAY LONGINT($tLon_Command_IDs;0)
ARRAY TEXT($tTxt_commands;0)

Repeat
   $Lon_id:=$Lon_id+1
   $Txt_command:=Command name($Lon_id)
   If(OK=1) //command number exists
      If(Length($Txt_command)>0) //command is not disabled
         APPEND TO ARRAY($tTxt_commands;$Txt_command)
         APPEND TO ARRAY($tLon_Command_IDs;$Lon_id)
      End if
   End if
Until(OK=0) //end of existing commands
```

## Example 2

In a form, you want a drop-down list populated with the basic summary report commands. In the object method for that drop-down list, you write:

```
Case of
   :(Form event=On Before)
      ARRAY TEXT(asCommand;4)
      asCommand{1}:=Command name(1)
      asCommand{2}:=Command name(2)
      asCommand{3}:=Command name(4)
      asCommand{4}:=Command name(3)
 `...
End case
```

In the English version of 4D, the drop-down list will read: Sum, Average, Min, and Max. In the French version*, the drop-down list will read: Somme, Moyenne, Min, and Max.

*with a 4D application configured to use the French programming language (see compatibility note)

## Example 3

You want to create a method that returns **True** if the command, whose number is passed as parameter, is thread-safe, and **False** otherwise.

```
 //Is_Thread_Safe project method
 //Is_Thread_Safe(numCom) -> Boolean

C_LONGINT($1;$threadsafe)
C_TEXT($name)
C_BOOLEAN($0)
$name:=Command name($1;$threadsafe;$theme)
If($threadsafe ?? 0) //if the first bit is set to 1
   $0:=True
Else
   $0:=False
End if
```

Then, for the "SAVE RECORD" command (53) for example, you can write:

```
$isSafe:=Is_Thread_Safe(53)
 // returns True
```

# Count parameters

Count parameters -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | | Number of parameters actually passed |

## Description

The **Count parameters** command returns the number of parameters passed to a project method.

**WARNING: Count parameters** is meaningful only in a project method that has been called by another method (project method or other). If the project method calling this command is associated with a menu, it returns 0.

## Example 1

4D project methods accept optional parameters, starting from the right.
For example, you can call the method **MyMethod(a;b;c;d)** in the following ways:

```
MyMethod(a;b;c;d) ` All parameters are passed
MyMethod(a;b;c) ` The last parameter is not passed
MyMethod(a;b) ` The last two parameters are not passed
MyMethod(a) ` Only the first parameter is passed
MyMethod ` No Parameter is passed at all
```

Using **Count parameters** from within **MyMethod,** you can detect the actual number of parameters and perform different operations depending on what you have received. The following example displays a text message and can insert the text into a 4D Write area or send the text into a document on disk:

```
  ` APPEND TEXT Project Method
  ` APPEND TEXT ( Text { ; Long { ; Time } } )
  ` APPEND TEXT ( Text { ; 4D Write Area { ; DocRef } } )

C_TEXT($1)
C_TIME($2)
C_LONGINT($3)

MESSAGE($1)
If(Count parameters>=3)
   SEND PACKET($3;$1)
Else
   If(Count parameters>=2)
      WR INSERT TEXT($2;$1)
   End if
End if
```

After this project method has been added to your application, you can write:

```
APPEND TEXT(vtSomeText) ` Will only display the text message
APPEND TEXT(vtSomeText;$wrArea) ` Displays text message and appends it to $wrArea
APPEND TEXT(vtSomeText;0;$vhDocRef) ` Displays text message and writes it to $vhDocRef
```

## Example 2

4D project methods accept a variable number of parameters of the same type, starting from the right. To declare these parameters, you use a compiler directive to which you pass *${N}* as a variable, where **N** specifies the first parameter. Using **Count parameters** you can address those parameters with a For loop and the parameter indirection syntax. This example is a function that returns the greatest number received as parameter:

```
  ` Max of Project Method
  ` Max of ( Real { ; Real2... ; RealN } ) -> Real
```

```
` Max of ( Value { ; Value2... ; ValueN } ) -> Greatest value

C_REAL($0;${1}) ` All parameters will be of type REAL as well as the function result
$0:=${1}
For($vlParam;2;Count parameters)
   If(${$vlParam}>$0)
      $0:=${$vlParam}
   End if
End for
```

After this project method has been added to your application, you can write:

```
vrResult:=Max of(Records in set("Operation A");Records in set("Operation B"))
```

or:

```
vrResult:=Max of(r1;r2;r3;r4;r5;r6)
```

# Current method name

Current method name -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String | | Calling method name |

## Description

The **Current method name** command returns the method name where it has been invoked. This command is useful for debugging generic methods.

According to the calling method type, the returned string can be as follows:

| Calling Method | Returned string |
|---|---|
| Database Method | MethodName |
| Trigger | Trigger on [TableName] |
| Project Method | MethodName |
| Table Form Method | [TableName].FormName |
| Project Form Method | FormName |
| Table Form Object Method | [TableName].FormName.ObjectName |
| Project Form Object Method | FormName.ObjectName |
| Component Project Method | MethodName |
| Component Project Form Method | FormName(ComponentName) |
| Component Project Form Object Method | FormName(ComponentName).ObjectName(ComponentName) |

This command cannot be called from within a 4D formula.

**Note:** For this command to be able to operate in compiled mode, the database must have been compiled with the **Range Checking** option (located in the Database Settings) selected.
In order to deactivate range checking in a method (or a part of a method) locally, you can use the following special comments:

```
`%R- to deactivate range checking
`%R+ to activate range checking
`%R* to restore the initial state of range checking (defined in the Preferences).
```

# EXECUTE METHOD

EXECUTE METHOD ( methodName {; result {; param}}{; param2 ; ... ; paramN} )

| Parameter | Type | | Description |
|---|---|---|---|
| methodName | String | ⇒ | Name of project method to be executed |
| result | Variable, Operator | ⇐ | Variable receiving the method result or * for a method not returning a result |
| param | Expression | ⇒ | Parameter(s) of the method |

## Description

The **EXECUTE METHOD** command executes the *methodName* project method while passing any parameters in *param1...paramN*. You can pass the name of any method that can be called from the database or the component executing the command.

In *result*, you can pass a variable which will receive the result of the execution of *methodName* (value placed in $0 inside *methodName*). If the method does not return a result, pass * as the second parameter. If the method does not return a result and does not require any parameters to be passed, pass only the *methodName* parameter.

The execution context is preserved in the called method, which means that the current form and any current form event remain defined.

If you call this command from a component and pass a method name belonging to the host database in *methodName* (or vice versa), the method must have been shared (“Shared by components and host database” option, in the Method properties).

## System variables and sets

If this command is executed correctly, the system variable OK is set to 1; otherwise, it is set to 0.

# EXECUTE METHOD IN SUBFORM

EXECUTE METHOD IN SUBFORM ( subformObject ; methodName {; return {; param} {; param2 ; ... ; paramN}} )

| Parameter | Type | | Description |
|---|---|---|---|
| subformObject | Text | ⇒ | Name of subform object |
| methodName | Text | ⇒ | Name of project method to be executed |
| return | Operator, Variable | ⇒ | * if method does not return a value |
| | | ⇐ | Value returned by method |
| param | Variable | ⇒ | Parameter(s) to pass to method |

## Description

The **EXECUTE METHOD IN SUBFORM** command can be used to execute the *methodName* project method in the context of the *subformObject* subform object.

The project method called can receive from 1 to X parameters in *param* and return a value in *return*. Pass * in the *return* parameter if the method does not return parameters.

You can pass the name of any project method that is accessible from the database or the component executing the command in *methodName*. The execution context is preserved in the method called, which means that the current form and current form event remain specified. If the subform comes from a component, the method must belong to the component and have the "Shared by components and host database" property.

This command must be called in the context of the parent form (containing the *subformObject* object), for example via the form method.

**Note:** The *methodName* method is not executed if the *subformObject* is not found in the current page or is not yet instantiated.

## Example 1

Given the "ContactDetail" form used as subform in the parent form "Company". The subform object that contains the ContactDetail form is named "ContactSubform". Imagine that we want to modify the appearance of certain elements of the subform according to the value of the field(s) of the company (for example, "contactname" must switch to red when [Company]City="New York" and to blue when [Company]City="San Diego"). This mechanism is implemented via the ***SetToColor*** method. To be able to get this result, the ***SetToColor*** method cannot be called directly from the process of the "On Load" form event of the Company parent form because the "contactname" object does not belong to the current form, but to the form displayed in the "ContactSubform" subform object. The method must therefore be executed using the **EXECUTE METHOD IN SUBFORM** command in order to function correctly.

```
Case of
   :(Form event=On Load)
      Case of
         :([Company]City="New York")
            $Color:=$Red
         :([Company]City="San Diego")
            $Color:=$Blue
         Else
            $Color:=$Black
      End case
      EXECUTE METHOD IN SUBFORM("ContactSubform";"SetToColor";*;$Color)
End case
```

## Example 2

You are developing a database that will be used as a component. It includes a shared project form (named, for instance, Calendar) that contains *dynamic variables* as well as a public project method that is used to adjust the calendar: ***SetCalendarDate(varDate)***.
If this method was used directly in the Calendar form method, you could call it directly in the "On Load" event:

```
SetCalendarDate(Current date)
```

But, in the context of the host database, imagine that a project form contains two "Calendar" subforms, in subform objects called "Cal1" and "Cal2". You must set the date of Cal1 to 01/01/10 and the date of Cal2 to 05/05/10. You cannot call ***SetCalendarDate*** directly because the method will not "know" which forms and variables it should apply. Therefore, you must call it via the following code:

```
EXECUTE METHOD IN SUBFORM("Cal1";"SetCalendarDate";*;!01/01/10!)
EXECUTE METHOD IN SUBFORM("Cal2";"SetCalendarDate";*;!05/05/10!)
```

## Example 3

Advanced example: in the same context as previously, this example provides a generic method:

```
 // Contents of the SetCalendarDate method
C_DATE($1)
C_TEXT($2)
Case of
   :(Count parameters=1)
 // Standard execution of method (as if it was executed from
 // the form itself) or specifically for a context (see case 2)

   :(Count parameters=2)
 // External call, needs a context
 // Recursive call with only one parameter
      EXECUTE METHOD IN SUBFORM($2;"SetCalendarDate";*;$1)
End case
```

## System variables and sets

If this command is executed correctly, the system variable OK is set to 1; otherwise, it is set to 0.

# Get pointer

Get pointer ( varName ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| varName | String | ⇒ | Name of a process or interprocess variable |
| Function result | Pointer | ⇐ | Pointer to process or interprocess variable |

## Description

The **Get pointer** command returns a pointer to the process or interprocess variable whose name you pass in *varName*.

To get a pointer to a field, use **Field**. To get a pointer to a table, use **Table**.

**Note:** You can pass expressions such as, for example, *ArrName+"{3}"*, as well as 2D array elements (*ArrName+"{3}{5}"*) to **Get pointer**.
However, you cannot pass variable elements (*ArrName+"{myVar}"*).

## Example 1

In a form, you build a 5 x 10 grid of enterable variables named v1, v2... v50. To initialize all of these variables, you write:

```
  ` ...
For($vlVar;1;50)
   $vpVar:=Get pointer("v"+String($vlVar))
   $vpVar->:=""
End for
```

## Example 2

Using pointers to elements of two-dimensional arrays:

```
$pt:=Get pointer("a{1}{2}")
 //$pt=->a{1}{2}
$pt2:=Get pointer("atCities"+"{2}{6}")
 //$pt2=->atCities{2}{6}
```

# Is a variable

Is a variable ( aPointer ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aPointer | Pointer | ⇒ | Pointer to be tested |
| Function result | Boolean | ⇐ | TRUE = Pointer points to a variable FALSE = Pointer does not point to a variable |

## Description

The **Is a variable** command returns True if the pointer you pass in *aPointer* references a defined variable. It returns False in all other cases (pointer to field or table, Nil pointer, and so on).

When you want to know the name of the variable being pointed to or the field number, you can use the **RESOLVE POINTER** command.

# Nil

Nil ( aPointer ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aPointer | Pointer | ⇒ | Pointer to be tested |
| Function result | Boolean | ⇐ | TRUE = Nil pointer (->[]) FALSE = Valid pointer to an existing object |

## Description

The **Nil** command returns True if the pointer you pass in *aPointer* is Nil (*->[]*). It returns False in all other cases (pointer to field, table or variable).

If you want to find out the name of the variable or the number of the field that is being pointed to, you can use the **RESOLVE POINTER** command.

# NO TRACE

NO TRACE
Does not require any parameters

## Description

You use **NO TRACE** when checking the execution of methods during the development of a database.

**NO TRACE** turns off the debugger engaged by **TRACE**, by an error, or by the user. Using **NO TRACE** has the same effect as clicking the No Trace button in the debugger.

In compiled databases, the **NO TRACE** command is ignored.

# RESOLVE POINTER

RESOLVE POINTER ( aPointer ; varName ; tableNum ; fieldNum )

| Parameter | Type | | Description |
|---|---|---|---|
| aPointer | Pointer | ⇒ | Pointer for which to retrieve the referenced object |
| varName | String | ⇐ | Name of referenced variable or empty string |
| tableNum | Longint | ⇐ | Number of referenced table or array element or 0 or -1 |
| fieldNum | Longint | ⇐ | Number of referenced field or 0 |

## Description

The **RESOLVE POINTER** command retrieves the information of the object referenced by the pointer expression *aPointer* and returns it into the parameters *varName*, *tableNum,* and *fieldNum*.

Depending on the nature of the referenced object, **RESOLVE POINTER** returns the following values:

| Referenced object | Parameters | | |
|---|---|---|---|
| | varName | tableNum | fieldNum |
| None (NIL pointer) | "" (empty string) | 0 | 0 |
| Variable | Name of the variable | -1 | -1 |
| Array | Name of the array | -1 | -1 |
| Array element | Name of the array | Element number | -1 |
| 2D array element | Name of the 2D array | Element row number | Element column number |
| Table | "" (empty string) | Table number | 0 |
| Field | "" (empty string) | Table number | Field number |

**Notes:**

- If the value you pass in *pointer* is not a pointer expression, a syntax error occurs.
- The **RESOLVE POINTER** command does not work with pointers to local variables. In fact, by definition several local variables with the same name could exist in different locations, so it is not possible for the command to find the correct variable.

## Example 1

Within a form, you create a group of 100 enterable variables called v1, v2... v100. To do so, you perform the following steps:

a. Create one enterable variable that you name v.

b. Set the properties of the object.

c. Attach the following method to that object:

```
DoSomething(Self) ` DoSomething being a project method in your database
```

d. At this point, you can either duplicate the variable as many times as you need, or use the Objects on Grid feature in the Form Editor.

e. Within the **DoSomething** method, if you need to know the index of the variable for which the method is called, you write:

```
RESOLVE POINTER($1;$vsVarName;$vlTableNum;$vlFieldNum)
$vlVarNum:=Num(Substring($vsVarName;2))
```

Note that by constructing your form in this way, you write the methods for the 100 variables only once; you do not need to write **DoSomething (1), DoSomething (2)...,DoSomething (100)**.

## Example 2

For debugging purposes, you need to verify that the second parameter ($2) to a method is a pointer to a table. At the beginning of this method, you write:

```
  ` ...
If(◊DebugOn)
   RESOLVE POINTER($2;$vsVarName;$vlTableNum;$vlFieldNum)
   If(Not(($vlTableNum>0)&($vlFieldNum=0)&($vsVarName="")))
  ` WARNING: The pointer is not a reference to a table
      TRACE
      End
   End if
  ` ...
```

## Example 3

See example for the **DRAG AND DROP PROPERTIES** command.

## Example 4

Here is an example of a 2D array pointer:

```
ARRAY TEXT(atCities;100;50)
C_POINTER($city)
atCities{1}{2}:="Rome"
atCities{1}{5}:="Paris"
atCities{2}{6}:="New York"
 // ...other values
$city:=->atCities{1}{5}
RESOLVE POINTER($city;$var;$rowNum;$colNum)
 //$var="atCities"
 //$rowNum="1"
 //$colNum="5"
```

# Self

Self -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Pointer | Pointer to form object (if any) whose method is currently being executed. Otherwise Nil (->[]) if outside of context |

## Compatibility Note

This command is kept only for compatibility reasons. Starting with version 12 of 4D, it is recommended to use the **OBJECT Get pointer** command.

## Description

The **Self** command returns a pointer to the object whose object method is currently being executed.

**Self** is used to reference a variable within its own object method. It returns a valid pointer when it is called from within an object method or from within a project method that is called directly or indirectly by an object method.

If **Self** is called out of context, it returns a **Nil** pointer (*->[]*).

**Tip: Self** is useful when several objects on a form need to perform the same task, yet operate on themselves.

**Note:** When it is used in the context of a list box, the function returns a pointer to the list box or the column of the list box depending on the context. For more information, please refer to the **Managing List Box Objects** section.

## Example

See the example for the **RESOLVE POINTER** command.

# TRACE

TRACE
Does not require any parameters

## Description

You use **TRACE** to trace methods during the development of a database.

The **TRACE** command turns on the 4D debugger for the current process. The **Debugger** window is displayed before the next line of code is executed, and continues to be displayed for each line of code that is executed. You can also turn on the debugger by pressing **Alt+Shift+right-click** (Windows) or **Control+Option+Command+click** (Macintosh) while code is executing.

In compiled databases, the **TRACE** command is ignored.

**4D Server:** If you call **TRACE** from a project method executed within the context of a Stored Procedure, the debugger window appears on the Server machine.

**Tip:** Do not place **TRACE** calls when using a form whose On Activate and On Deactivate events have been enabled. Each time the debugger window appears, these events will be invoked; you will then loop infinitely between these events and the debugger window. If you end up in this situation, **Shift+click** on the **No Trace** button of the debugger in order to get out of it. Any subsequent calls to **TRACE** within the process will be ignored.

## Example

The following code expects the process variable **BUILD_LANG** to be equal to “US” or “FR”. If this is not the case, it calls the project method **DEBUG**:

```
  ` ...
Case of
   :(BUILD_LANG="US")
      vsBHCmdName:=[Commands]CM US Name
   :(BUILD_LANG="FR")
      vsBHCmdName:=[Commands]CM FR Name
   Else
      DEBUG("Unexpected BUILD_LANG value")
End case
```

The **DEBUG** project method is listed here:

```
  ` DEBUG Project Method
  ` DEBUG (Text)
  ` DEBUG (Optional Debug Information)

C_TEXT($1)

If(◊vbDebugOn) ` Interprocess variable set in the On Startup Method
   If(Is compiled mode)
      If(Count parameters>=1)
         ALERT($1+Char(13)+"Call Designer at x911")
      End if
   Else
      TRACE
   End if
End if
```

# Type

Type ( fieldVar ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| fieldVar | Field, Variable | ⇒ | Field or Variable to be tested |
| Function result | Longint | ⇐ | Data type number |

## Description

The **Type** command returns a numeric value that denotes the type of the field or variable you pass as *fieldVar*.

4D provides the following predefined constants found in the **Field and Variable Types** theme:

| Constant | Type | Value |
|---|---|---|
| Array 2D | Longint | 13 |
| Blob array | Longint | 31 |
| Boolean array | Longint | 22 |
| Date array | Longint | 17 |
| Integer array | Longint | 15 |
| Is alpha field | Longint | 0 |
| Is BLOB | Longint | 30 |
| Is Boolean | Longint | 6 |
| Is date | Longint | 4 |
| Is float | Longint | 35 |
| Is integer | Longint | 8 |
| Is integer 64 bits | Longint | 25 |
| Is JSON null | Longint | 255 |
| Is longint | Longint | 9 |
| Is object | Longint | 38 |
| Is picture | Longint | 3 |
| Is pointer | Longint | 23 |
| Is real | Longint | 1 |
| Is string var | Longint | 24 |
| Is subtable | Longint | 7 |
| Is text | Longint | 2 |
| Is time | Longint | 11 |
| Is undefined | Longint | 5 |
| LongInt array | Longint | 16 |
| Object array | Longint | 39 |
| Picture array | Longint | 19 |
| Pointer array | Longint | 20 |
| Real array | Longint | 14 |
| String array | Longint | 21 |
| Text array | Longint | 18 |
| Time array | Longint | 32 |

**Notes**:

- **Type** returns *9* (Is longint) when applied to a Graph variable.
- Beginning with version 11 of 4D, **Type** returns the actual type of an array when it is applied to a "row" of a 2D array, rather than Array 2D as before (see example 4).
- Beginning with version 11 of 4D, **Type** returns the type Is text or Text array when it is applied, respectively, to an Alpha variable or an Alpha array (starting with this version, there is no difference between an Alpha variable and a Text variable).

You can apply **Type** to fields, interprocess variables, process variables, local variables, and dereferenced pointers referring to these types of objects.You can apply **Type** to parameters (*$1,$2..., ${...}*), or to project method or function results (*$0*).

## Example 1

See example for the **APPEND DATA TO PASTEBOARD** command.

## Example 2

See example for **DRAG AND DROP PROPERTIES** command.

## Example 3

The following project method empties some or all of the fields for the current record of the table whose a pointer is passed as parameter. It does this without deleting or changing the current record:

```
  ` EMPTY RECORD Project Method
  ` EMPTY RECORD ( Pointer {; Long } )
  ` EMPTY RECORD ( -> [Table] { ; Type Flags } )

C_POINTER($1)
C_LONGINT($2;$vlTypeFlags)

If(Count parameters>=2)
   $vlTypeFlags:=$2
Else
   $vlTypeFlags:=0xFFFFFFFF
End if
For($vlField;1;Get last field number($1))
   $vpField:=Field(Table($1);$vlField)
   $vlFieldType:=Type($vpField->)
   If($vlTypeFlags ??$vlFieldType )
      Case of
         :(($vlFieldType=Is alpha field)|($vlFieldType=Is text))
            $vpField->:=""
         :(($vlFieldType=Is real)|($vlFieldType=Is integer)|($vlFieldType=Is longint))
            $vpField->:=0
         :($vlFieldType=Is date)
            $vpField->:=!00/00/00!
         :($vlFieldType=Is time)
            $vpField->:=?00:00:00?
         :($vlFieldType=Is Boolean)
            $vpField->:=False
         :($vlFieldType=Is picture)
            C_PICTURE($vgEmptyPicture)
            $vpField->:=$vgEmptyPicture
         :($vlFieldType=Is subtable)
            Repeat
               ALL SUBRECORDS($vpField->)
               DELETE SUBRECORD($vpField->)
            Until(Records in subselection($vpField->)=0)
         :($vlFieldType=Is BLOB)
            SET BLOB SIZE($vpField->;0)
      End case
   End if
End for
```

After this project method is implemented in your database, you can write:

```
  ` Empty the whole current record of the table [Things To Do]
EMPTY RECORD(->[Things To Do])

  ` Empty Text, BLOB and Picture fields for the current record of the table [Things To Do]
EMPTY RECORD(->[Things To Do];0?+Is text?+Is BLOB?+Is picture)
```

```
` Empty the whole current record of the table [Things To Do] except Alphanumeric fields
EMPTY RECORD(->[Things To Do];-1?-Is alpha field)
```

## Example 4

In certain cases, for example when writing generic code, you may need to find out whether an array is a standard independent array or the “row” of a 2D array. In this case, you can use the following code:

```
ptrmyArr:=->myArr{6} ` Is myArr{6} the row of a 2D array?
RESOLVE POINTER(ptrmyArr;varName;tableNum;fieldNum)
If(varName#"")
   $ptr:=Get pointer(varName)
   $thetype:=Type($ptr->)
 ` If myArr{6} is the row of a 2D array, $thetype equals 13
End if
```

# LDAP


- Overview of LDAP commands
- LDAP LOGIN
- LDAP LOGOUT
- LDAP Search
- LDAP SEARCH ALL

## Overview of LDAP commands

The commands of the "LDAP" theme allow your 4D application to connect to a company directory server such as MS Active Directory using the LDAP protocol. You can access the server data and query it.

**Note:** LDAP or Lightweight Directory Access Protocol is an open standard for accessing and maintaining distributed information services. For more information, please refer to the Wikipedia page on LDAP or the OpenLDAP Software main page.

LDAP commands allow you to:

- use the Windows session login and password to grant access to your 4D application (if using MS Active directory) so that the end user only needs to remember a single password,
- query the corporate directory to retrieve user information such as full name, email, phone number, office building, groups the person belongs to, etc.

In 4D, an LDAP connection is opened using **LDAP LOGIN**. It is then bound to the current 4D process and must be closed using **LDAP LOGOUT**, or when the process terminates its execution.

### Glossary

Here is a list of the main acronyms used in the LDAP environment:

| Acronym | Definition |
|---|---|
| LDAP | Lightweight Directory Access Protocol |
| AD | Active Directory. AD is a directory services database implemented by Microsoft, and LDAP is one of the protocols you can use to talk to it. |
| CN | Common Name, for example "John Doe" |
| DN | Distinguished Name, for example "cn=John Doe,ou=users,dc=example,dc=com" |
| SAM-Account-Name | Security Account Manager, logon name for AD, for example "jdoe" |
| OU | Organizational unit, groups of the server tree |
| DC | Domain components, root and first branches of the server tree |
| uid | User identifier |

# LDAP LOGIN

LDAP LOGIN ( url ; login ; password {; digest} )

| Parameter | Type | | Description |
|---|---|---|---|
| url | String | ⇒ | URL of LDAP server to connect to |
| login | String | ⇒ | Login entry |
| password | String | ⇒ | Password of login entry |
| digest | Longint | ⇒ | 0 = send password in digest MD5 (default), 1 = send password without encryption |

## Description

The **LDAP LOGIN** command opens a read-only connection to the LDAP server specified in the *url* parameter with the *login* and *password* identifiers provided. If accepted by the server, this connection will be used for any LDAP searches executed subsequently in the current process until the **LDAP LOGOUT** command is executed (or until the process is closed).

In *url*, pass the full URL of the LDAP server you want to connect to, including the scheme and port (389 by default). This parameter should be compliant with rfc2255.
You can open secure connections over TLS by using a *url* that starts with "ldaps" and uses a specific port number (for example "ldaps://svr.ldap.acme.com:1389"). The LDAP server must have an SSL Certificate (at least for Microsoft Active Directory). Using a TLS connection is highly recommended when the password is sent in plain text (see below).

**Note:** If you pass an empty string in the *url* parameter, the command will try to connect to the default LDAP server available on the domain; (this feature is intended for testing purposes only, for performance reasons it should not be used in production).

In *login*, pass the user account on the LDAP server, and in *password*, pass the user password. By default, the *login* can be one of the following login strings, depending on the LDAP Server configuration:

- a Distinguished Name (DN), for example "CN=John Smith,OU=users,DC=example,DC=com"
- the user name (CN), for example "CN=John Smith"
- an e-mail address, for example "johnsmith@4d.fr"
- an SAM-Account-Name, for example "jsmith".

Note that accepted values for the *login* are related to the password transmission mode as defined by the *digest* parameter. For example, in a default MS Active Directory configuration:

- When the transmission mode is LDAP password MD5, the only accepted value for a login is the SAM-Account-Name.
- When the transmission mode is LDAP password plain text (clear text), the *login* parameter can be either DN, CN or an e-mail address. An SAM-Account-Name is also accepted but only when preceded by the domain name (for example, "dc-acme.com/jsmith").

The *digest* parameter allows you to modify the way the password is transmitted over the network. You can use one of the following constants, located in the "**LDAP**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| LDAP password MD5 | Longint | 0 | (Default) Send password encrypted in MD5 |
| LDAP password plain text | Longint | 1 | Send password with no encryption (TLS connection recommended) |

By default, the *password* is transmitted in digest MD5. Pass LDAP password plain text if necessary, for example if you want to use different login type values with the LDAP server. In a production environment, it is recommended to use a TLS connection for the *url*.

**Note:** Authentication with an empty password lets you enter the anonymous binding mode (if authorized by the LDAP server). However, in this mode, errors can be thrown if you try to perform any operation that is not allowed in this kind of bind.

If the login parameters are valid, a connection to the LDAP server is opened in the 4D process. You can then search and retrieve information using LDAP commands.

Do not forget to call the **LDAP LOGOUT** command when the connection to the LDAP server is no longer necessary.

## Example 1

You want to log in to an LDAP server and do a search:

```
ARRAY TEXT($_tabAttributes;0)
APPEND TO ARRAY($_tabAttributes;"cn")
APPEND TO ARRAY($_tabAttributes;"phoneNumber")
LDAP LOGIN("ldap://srv.dc.acme.com:389";"John Smith";"qrnSurBret2elburg")
$vfound:=LDAP Search("OU=UO_Users,DC=ACME,DC=com";cn=John Doe";LDAP all levels;$_tabAttributes)
LDAP LOGOUT //do not forget to log out
```

## Example 2

This example tries to connect to an application:

```
ON ERR CALL("ErrHdlr") //handle errors
errOccurred:=False
errMsg:=""
If(ppBindMode=1) //if password mode is default
   LDAP LOGIN(vUrlLdap;vUserCN;vPwd;LDAP password MD5)
Else
   LDAP LOGIN(vUrlLdap;vUserCN;vPwd;LDAP password plain text)
End if

Case of
   :(Not(errOccurred))
      ALERT(" You are now connected to your LDAP server.  ")

   :(errOccurred)
      ALERT("Error in your parameters")
End case

LDAP LOGOUT
ON ERR CALL("")
```

# LDAP LOGOUT

LDAP LOGOUT
Does not require any parameters

## Description

The **LDAP LOGOUT** command closes the connection with an LDAP server in the current process (if applicable). If there is no connection, an error 1003 stating that you are not logged in is returned.

# LDAP Search

LDAP Search ( dnRootEntry ; filter {; scope {; attributes {; attributesAsArray}}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| dnRootEntry | String | ➡ | Distinguished Name of root entry where search is to start |
| filter | String | ➡ | LDAP search filter |
| scope | String | ➡ | Scope of search: "base" (default), "one", or "sub" |
| attributes | Text array | ➡ | Attribute(s) to fetch |
| attributesAsArray | Boolean array | ➡ | True = force attributes to be returned as array; False = force attributes to be returned as a simple variable |
| Function result | Object | ⮌ | Key/value attributes |

## Description

The **LDAP Search** command searches in the target LDAP server for the first occurrence matching the criteria defined. This command must be executed within a connection to an LDAP server opened with **LDAP LOGIN**; otherwise a 1003 error is returned.

In *dnRootEntry*, pass the *Distinguished Name* of the LDAP server root entry; the search will start at this entry.

In *filter*, pass the LDAP search filter to execute. The filter string must be compliant with rfc2225. You can pass an empty string "" in order not to filter the search; the "*" is supported to search substrings.

In *scope*, pass one of the following constants from the "**LDAP**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| LDAP all levels | String | sub | Search in the root entry level defined by *dnRootEntry* and in all subsequent entries |
| LDAP root and next | String | one | Search in the root entry level defined by *dnRootEntry* and in the directly subsequent entries on one level |
| LDAP root only | String | base | Search only in the root entry level defined by *dnRootEntry* (default if omitted) |

In *attributes*, pass a text array which contains the list of all LDAP attributes to fetch from the matched entries. By default, if this parameter is omitted, all attributes are fetched.

**Note:** Keep in mind that LDAP attribute names are case-sensitive. For more information on LDAP attributes, you can refer to this page that lists all available attributes for the MS Active directory.

By default, the command returns attributes as an array if multiple results are found, or as a variable if a single result is found. The optional *attributesAsArray* parameter allows you to "force" returned attribute(s) to be formatted as an array or as a variable for each attribute defined:

- When you pass **true** in an element, the corresponding element of the *attributes* parameter will be returned in an array. If a single value is found, the command returns an array with a single element.
- When you pass **false** in an element, the corresponding element of the *attributes* parameter will be returned in a simple variable. If multiple entries are found, the command returns only the first element.

## Example 1

You want to get the phone number of the user "smith" in the company directory:

```
ARRAY TEXT($_tabAttributes;0)
APPEND TO ARRAY($_tabAttributes;"cn")
APPEND TO ARRAY($_tabAttributes;"phoneNumber")
LDAP LOGIN($url;$dn;$pwd)
$filter:="cn=*smith*"
$vfound:=LDAP Search($dnSearchRootEntry;$filter;LDAP all levels;$_tabAttributes)
LDAP LOGOUT
```

## Example 2

We want to get an array of all entries found for the "memberOf" attribute:

```
C_OBJECT($entry)
ARRAY TEXT($_tabAttributes;0)
ARRAY BOOLEAN($_tabAttributes_asArray;0)
APPEND TO ARRAY($_tabAttributes;"cn")
APPEND TO ARRAY($_tabAttributes_asArray;False)
APPEND TO ARRAY($_tabAttributes;"memberOf")
APPEND TO ARRAY($_tabAttributes_asArray;True)

LDAP LOGIN($url;$login;$pwd;LDAP password plain text)
$entry:=LDAP Search($dnSearchRootEntry;"cn=adrien*";LDAP all levels;$_tabAttributes;$_tabAttributes_asArray)
LDAP LOGOUT

ARRAY TEXT($_arrMemberOf;0)
OB GET ARRAY($entry;"memberOf";$_arrMemberOf)
 // in $_arrMemberOf we have an array containing all entry groups
```

# LDAP SEARCH ALL

LDAP SEARCH ALL ( dnRootEntry ; arrResult ; filter {; scope {; attributes {; attributesAsArray}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| dnRootEntry | String | ⇒ | Distinguished Name of root entry where search is to start |
| arrResult | Object array | ⇐ | Result of the search |
| filter | String | ⇒ | LDAP search filter |
| scope | String | ⇒ | Scope of the search: "base" (default), "one", or "sub" |
| attributes | Text array | ⇒ | Attribute(s) to fetch |
| attributesAsArray | Boolean array | ⇒ | True = force attributes to be returned as array; false = force attributes to be returned as a simple variable |

## Description

The **LDAP SEARCH ALL** command searches in the target LDAP server for all occurrences matching the criteria defined. This command must be executed within a connection to an LDAP server opened with **LDAP LOGIN**; otherwise a 1003 error is returned.

Note that LDAP servers usually impose a maximum number of entries that can be received from a search. For example, Microsoft Active directory limits this number to 1000 entries by default.

In *dnRootEntry*, pass the *Distinguished Name* of the LDAP server root entry; the search will start at this entry.

In *arrResult*, pass an object array that will be filled with all matching entries; in this array, each element is an object containing attribute/value pairs returned for a matching entry. You can use the *attributes* parameter to define the attributes to be returned.

In *filter*, pass the LDAP search filter to execute. The filter string must be compliant with rfc2225. You can pass an empty string "" in order not to filter the search; the "*" is supported to search substrings.

In *scope*, pass one of the following constants from the "**LDAP**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| LDAP all levels | String | sub | Search in the root entry level defined by *dnRootEntry* and in all subsequent entries |
| LDAP root and next | String | one | Search in the root entry level defined by *dnRootEntry* and in the directly subsequent entries on one level |
| LDAP root only | String | base | Search only in the root entry level defined by *dnRootEntry* (default if omitted) |

In *attributes*, pass a text array which contains the list of all LDAP attributes to fetch from the matched entries. By default, if this parameter is omitted, all attributes are fetched.

**Note:** Keep in mind that LDAP attribute names are case-sensitive. For more information on LDAP attributes, you can refer to this page that lists all available attributes for the MS Active directory.

By default, the command returns attributes as an array if multiple results are found, or as a variable if a single result is found. The optional *attributesAsArray* parameter allows you to "force" the attribute(s) returned to be formatted as an array or as a variable for each attribute defined:

- When you pass **true** in an element, the corresponding element of the *attributes* parameter will be returned in an array. If a single value is found, the command returns an array with a single element.
- When you pass **false** in an element, the corresponding element of the *attributes* parameter will be returned in a simple variable. If multiple entries are found, the command returns only the first element.

## Example 1

We want to get the phone number of all users named "smith" in the company directory:

```
ARRAY TEXT($_tabAttributes;0)
ARRAY BOOLEAN($_tabAttributes_asArray;0)
APPEND TO ARRAY($_tabAttributes;"cn")
APPEND TO ARRAY($_tabAttributes_asArray;False)
APPEND TO ARRAY($_tabAttributes;"telephoneNumber")
APPEND TO ARRAY($_tabAttributes_asArray;False)
ARRAY OBJECT($_entry;0)
```

```
LDAP LOGIN($url;$myLogin;$pwd)
$filter:="cn=*smith*"
LDAP SEARCH ALL($dnSearchRootEntry;$_entry;$filter;LDAP all levels;$_tabAttributes)
LDAP LOGOUT


 //$_entry will contain for example
 // $_entry{1} = {"cn":"John Smith","telephoneNumber":"01 40 87 00 00"}
 // $_entry{2} = {"cn":"Adele Smith","telephoneNumber":"01 40 87 00 01"}
 // $_entry{3} = {"cn":"Adrian Smith","telephoneNumber":"01 23 45 67 89"}
 // ...
```

## Example 2

These examples illustrate the use of the *attributesAsArray* parameter:

```
ARRAY OBJECT($_entry;0)
ARRAY TEXT($_tabAttributes;0)
ARRAY BOOLEAN($_tabAttributes_asArray;0)
APPEND TO ARRAY($_tabAttributes;"cn")
APPEND TO ARRAY($_tabAttributes_asArray;False)
APPEND TO ARRAY($_tabAttributes;"memberOf")
APPEND TO ARRAY($_tabAttributes_asArray;True)

LDAP LOGIN($url;$login;$pwd;LDAP password plain text)
LDAP SEARCH ALL($dnSearchRootEntry;$_entry;$filter;LDAP all levels;$_tabAttributes;$_tabAttributes_asArray)
LDAP LOGOUT

ARRAY TEXT($_arrMemberOf;0)
OB GET ARRAY($_entry{1};"memberOf";$_arrMemberOf)
 // in $_arrMemberOf we have an array containing all groups of the entry
```

```
ARRAY TEXT($_tabAttributes;0)
ARRAY BOOLEAN($_tabAttributes_asArray;0)
APPEND TO ARRAY($_tabAttributes;"cn")
APPEND TO ARRAY($_tabAttributes_asArray;False)
APPEND TO ARRAY($_tabAttributes;"memberOf")
APPEND TO ARRAY($_tabAttributes_asArray;False)

LDAP LOGIN($url;$login;$pwd;LDAP password plain text)
LDAP SEARCH ALL($dnSearchRootEntry;$_entry;$filter;LDAP all levels;$_tabAttributes;$_tabAttributes_asArray)
LDAP LOGOUT

$memberOf:=OB Get($_entry{1};"memberOf")
 // in $memberOf we have a variable containing the first group of the entry
```

# List Box

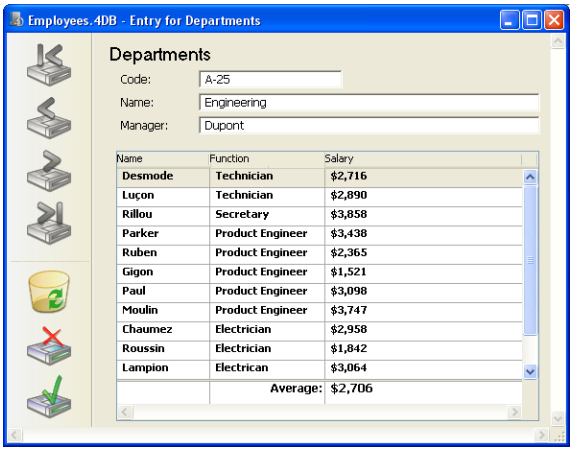
- Managing List Box Objects
- Managing Hierarchical List Boxes
- Using object arrays in columns (4D View Pro)
- 4D View Pro
- LISTBOX COLLAPSE
- LISTBOX DELETE COLUMN
- LISTBOX DELETE ROWS
- LISTBOX DUPLICATE COLUMN
- LISTBOX EXPAND
- LISTBOX Get array Updated 16.0
- LISTBOX GET ARRAYS
- LISTBOX GET CELL COORDINATES
- LISTBOX GET CELL POSITION
- LISTBOX Get column formula
- LISTBOX Get column width
- LISTBOX Get footer calculation
- LISTBOX Get footers height
- LISTBOX GET GRID
- LISTBOX GET GRID COLORS
- LISTBOX Get headers height
- LISTBOX GET HIERARCHY
- LISTBOX Get information
- LISTBOX Get locked columns
- LISTBOX Get number of columns
- LISTBOX Get number of rows
- LISTBOX GET OBJECTS
- LISTBOX GET PRINT INFORMATION
- LISTBOX Get row color
- LISTBOX Get row font style
- LISTBOX Get row height New 16.0
- LISTBOX Get rows height
- LISTBOX Get static columns
- LISTBOX GET TABLE SOURCE
- LISTBOX INSERT COLUMN
- LISTBOX INSERT COLUMN FORMULA
- LISTBOX INSERT ROWS
- LISTBOX MOVE COLUMN
- LISTBOX MOVED COLUMN NUMBER
- LISTBOX MOVED ROW NUMBER
- LISTBOX SELECT BREAK
- LISTBOX SELECT ROW
- LISTBOX SET ARRAY Updated 16.0
- LISTBOX SET COLUMN FORMULA
- LISTBOX SET COLUMN WIDTH
- LISTBOX SET FOOTER CALCULATION
- LISTBOX SET FOOTERS HEIGHT
- LISTBOX SET GRID
- LISTBOX SET GRID COLOR
- LISTBOX SET HEADERS HEIGHT
- LISTBOX SET HIERARCHY
- LISTBOX SET LOCKED COLUMNS
- LISTBOX SET ROW COLOR

- LISTBOX SET ROW FONT STYLE
- LISTBOX SET ROW HEIGHT New 16.0
- LISTBOX SET ROWS HEIGHT
- LISTBOX SET STATIC COLUMNS
- LISTBOX SET TABLE SOURCE
- LISTBOX SORT COLUMNS

# Managing List Box Objects

The commands of this theme are dedicated to handling form objects of the List box type.

List boxes represent data in the form of columns and selectable rows and provide many interface functions such as the ability to enter values, sort columns, display a hierarchy, define alternating colors, and so on.



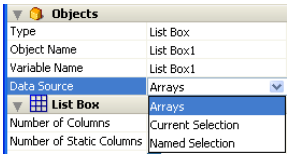

You can set up a List box completely in the 4D Form editor and can also manage it through programming. For more information on creating and setting List boxes in the Form editor as well as on their use, refer to the *Design Reference* manual of the 4D documentation.

Programming list box objects is done in the same way as other 4D list form objects. However, specific rules must be followed, as detailed in this section.

## Data sources and principles for managing values

A list box can contain one or more columns and can be associated either with 4D arrays or a selection of records. In the case of selection type list boxes, columns are associated with fields or expressions.

It is not possible to have both types of data sources (arrays and selections) combined in the same list box. The data source is set when the list box is created in the Form editor, via the Property list. It is then no longer possible to modify it by programming.




### Array type list boxes

In this type of list box, each column must be associated with a one-dimensional 4D array; all array types can be used, with the exception of pointer arrays. The display format for each column can be defined in the Form editor or by using the **OBJECT SET FORMAT** command.

Using the language, the values of columns (data entry and display) are managed using high-level List box commands (such as **LISTBOX INSERT ROWS** or **LISTBOX DELETE ROWS**) as well as array manipulation commands.
For example, to initialize the contents of a column, you can use the following instruction:

```
ARRAY TEXT(ColumnName;size)
```

You can also use a list:

```
LIST TO ARRAY("ListName";ColumnName)
```

**Warning** : When a list box contains several columns of different sizes, only the number of items of the smallest array (column) will be displayed. You should make sure that each array has the same number of elements as the others. Also, if a

list box column is empty (this occurs when the associated array was not correctly declared or sized using the language), the list box displays nothing.

### Selection type list boxes

In this type of list box, each column can be associated with a field or an expression. The contents of each row is then evaluated according to a selection of records: the current selection of a table or a named selection.

When the current selection is the data source, any modifications made on the database side are automatically carried over to the list box and vice versa. The current selection is thus always the same in both locations. Note that the **LISTBOX INSERT ROWS** and **LISTBOX DELETE ROWS** commands cannot be used with selection type list boxes.

You can associate a list box column with an expression. The expression could be based on one or more fields (for example *[Employees]LastName+" "+[Employees]FirstName*) or simply be a formula (for example **String**(Milliseconds)). The expression can also be a project method, a variable or an array element.

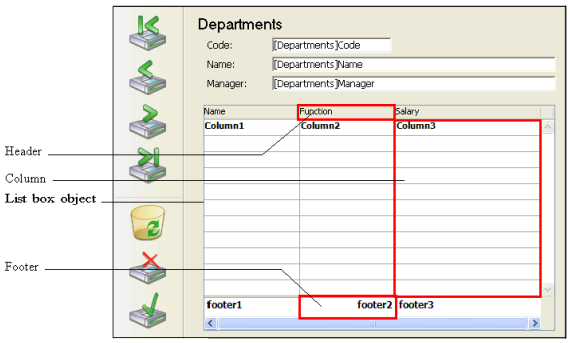
The **LISTBOX SET TABLE SOURCE** command can be used to modify the table associated with the list box by programming.

## Object, column, header and footer

A List box object is composed of four separate items:

- the **object** itself,
- the **columns**,
- the column **headers** (can be displayed or hidden, displayed by default),
- the column **footers** (can be displayed or hidden, hidden by default).

These items can be selected individually in the Form editor. Each one has its own object and variable name and can be handled separately.




By default, columns are named **Column1** to *n*, headers **Header1** to *n* and footers **Footer1** to *n* in the form, independently of the list box objects. Note that, by default, the same name is used for the objects and their associated variables, except in the case of footers (by default, footer variables are blank; 4D uses dynamic variables).

Each item type contains individual and shared characteristics with other items. For example, character fonts can be globally assigned to the list box object or separately to columns, headers and/or footers. On the other hand, entry properties can only be defined for columns.

These rules apply to the "**Object Properties**" theme commands that can be used with list boxes. Depending on its functionality, each command can be used with the list box, columns, column headers and/or footers. To set the type of item on which you want to work, simply pass the object name or the variable associated with it.

The following table details the scope of each command of the "**Object Properties**" theme that can be used with list boxes:

| Object Properties commands | Object | Columns | Column headers | Column footers |
|---|---|---|---|---|
| OBJECT MOVE | X | | | |
| OBJECT GET COORDINATES | X | X | X | X |
| OBJECT SET RESIZING OPTIONS | X | | | |
| OBJECT GET RESIZING OPTIONS | X | | | |
| OBJECT GET BEST SIZE | | X | | |
| OBJECT SET FILTER | | X | | |
| OBJECT SET FORMAT | | X | X | X |
| OBJECT SET ENTERABLE | | X | | |
| OBJECT SET LIST BY NAME | | X | | |
| OBJECT SET TITLE | | | X | |
| OBJECT SET COLOR | X | X | X | X |
| OBJECT SET RGB COLORS | X | X | X | X |
| OBJECT SET FONT | X | X | X | X |
| OBJECT SET FONT SIZE | X | X | X | X |
| OBJECT SET FONT STYLE | X | X | X | X |
| OBJECT SET HORIZONTAL ALIGNMENT | X | X | X | X |
| OBJECT Get horizontal alignment | X | X | X | X |
| OBJECT SET VERTICAL ALIGNMENT | X | X | X | X |
| OBJECT Get vertical alignment | X | X | X | X |
| OBJECT SET VISIBLE | X | X | X | X |
| OBJECT SET SCROLLBAR | X | | | |
| OBJECT SET SCROLL POSITION | X | | | |

**Note**: With array type list boxes, it is possible to specify the style, font color, background color and visibility of each row separately. This is managed via arrays associated with the list box in the Property List. You can retrieve the names of these arrays by programming using the **LISTBOX GET ARRAYS** command.

## List box and Language

### Object methods

It is possible to add an object method to the list box object and/or to each column of the list box. Object methods are called in the following order:

**1.** Object method of each column

**2.** Object method of the list box

The column object method gets events that occur in its header and footer.

### OBJECT SET VISIBLE and headers/footers

When the **OBJECT SET VISIBLE** command is used with a header or footer, it is used on all List box object headers or footers, regardless of the individual element set by the command. For example, the **OBJECT SET VISIBLE(*;"header3";False)** instruction will hide all headers in the List box object to which header3 belongs and not simply this header.

Note that in order for you to be able to manage the visibility of these objects using the **OBJECT SET VISIBLE** command, they must have been displayed in the list box at the level of the Form editor (the **Display Headers** and/or **Display Footers** option must be checked for the object).

### OBJECT Get pointer

The **OBJECT Get pointer** function used with the Object with focus or Object current constant (formerly the **Focus object** and **Self** functions) can be used in the object method of a list box or a list box column. They return a pointer to the list box, the list box column(1) or the header variable depending on the type of form event. The following table details this functioning:

| Event | Object with focus | Object current |
|---|---|---|
| On Clicked | list box | column |
| On Double Clicked | list box | column |
| On Before Keystroke | column | column |
| On After Keystroke | column | column |
| On After Edit | column | column |
| On Getting Focus | column or list box (*) | column or list box (*) |
| On Losing Focus | column or list box (*) | column or list box (*) |
| On Drop | list box source | list box (*) |
| On Drag Over | list box source | list box (*) |
| On Begin Drag Over | list box | list box (*) |
| On Mouse Enter | list box (**) | list box (**) |
| On Mouse Move | list box (**) | list box (**) |
| On Mouse Leave | list box (**) | list box (**) |
| On Data Change | column | column |
| On Selection Change | list box (**) | list box (**) |
| On Before Data Entry | column | column |
| On Column Moved | list box | column |
| On Row Moved | list box | list box |
| On Column Resize | list box | column |
| On Open Detail | Nil | list box (**) |
| On Close Detail | Nil | list box (**) |
| On Header Click | list box | header |
| On Footer Click | list box | footer |
| On After Sort | list box | header |

(*) When the focus is modified within a list box, a pointer to the column is returned. When the focus is modified at the overall form level, a pointer to the list box is returned. In the context of a column object method, a pointer to the column is returned.
(**) Not executed in the context of a column object method.

(1) When a pointer to a column is returned, the object pointed to depends on the type of list box. With an array type list box, the **OBJECT Get pointer** function (“User Interface” theme) returns a pointer to the column of the list box with the focus (i.e. to an array). The 4D pointer mechanism allows you to see the item number of the modified array. For example, supposing a user modified the 5th line of the column col2:

```
$Column:=OBJECT Get pointer(Object with focus)
` $Column contains a pointer to col2
$Row:=$Column-> `$Row equals 5
```

For a selection type list box, the **OBJECT Get pointer** function returns:

- For a column associated with a field, a pointer to the associated field,
- For a column associated with a variable, a pointer to the variable,
- For a column associated with an expression, the **Nil** pointer.

### OBJECT SET SCROLL POSITION

The **OBJECT SET SCROLL POSITION** command (“**Objects (Forms)**” theme) can be used with a list box. It scrolls the list box rows so that the first selected row or a specified row is displayed.

### EDIT ITEM

The **EDIT ITEM** command (“**Entry Control**” theme) allows you to pass a cell of a list box object into edit mode.

### REDRAW

When it is applied to a listbox in selection mode, the **REDRAW** command ("**User Interface**" theme) triggers the updating of the data displayed in the list box.

### Displayed line number

The **Displayed line number** command (“**Selection**” theme) functions in the context of the On Display Detail form event for a list box object.

### Form events

Specific form events are intended to facilitate list box management, in particular concerning drag and drop and sort operations. For more information, refer to the description of the **Form event** command.

### Drag and drop

Managing the drag and drop of data in list boxes is supported by the **Drop position** and **DRAG AND DROP PROPERTIES** commands. These commands have been specially adapted for list boxes.

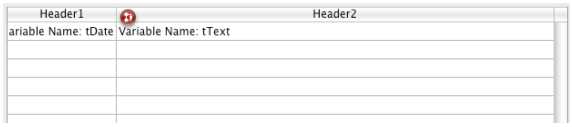
Be careful not to confuse drag and drop with the moving of rows and columns, supported by the **LISTBOX MOVED ROW NUMBER** and **LISTBOX MOVED COLUMN NUMBER** commands.

## Managing entry

For a list box cell to be enterable, both of the following conditions must be met:

- The cell’s column must have been set as **Enterable** (otherwise, the cells of the column can never be enterable).
- In the On Before Data Entry event, $0 does not return -1.
  When the cursor arrives in the cell, the On Before Data Entry event is generated in the column method. If, in the context of this event, $0 is set to -1, the cell is considered as not enterable. If the event was generated after **Tab** or **Shift+Tab** was pressed, the focus goes to either the next cell or the previous one respectively. If $0 is not -1 (by default $0 is 0), the cell is enterable and switches to editing mode.

Let’s consider the example of a list box containing two arrays, one date and one text. The date array is not enterable but the text array is enterable if the date has not already past.




Here is the method of the arrText column:

```
Case of
   :(Form event=On Before Data Entry) // a cell gets the focus
      LISTBOX GET CELL POSITION(*;"lb";$col;$row)
 // identification of cell
      If(arrDate{$row}<Current date) // if date is earlier than today
         $0:=-1 // cell is NOT enterable
      Else
 // otherwise, cell is enterable
      End if
End case
```

**Note:** Beginning with 4D v13, the On Before Data Entry event is returned before On Getting Focus.

In order to preserve data consistency for selection type list boxes, any modified record is saved as soon as the cell is validated (if it is set, the On saving an existing record trigger is called) then the On Data Change event is executed. The typical sequence of events generated during data entry or modification is as follows:

| Action | Sequence of events |
|---|---|
| A cell switches to edit mode | On Before Data Entry / On Getting Focus |
| Its value is modified | On Before Keystroke / On After Keystroke / On After Edit |
| A user validates and leaves the cell (tab, click, etc.) | *Save* (On saving an existing record trigger) / On Data Change / On Losing Focus |

## Managing sorts

By default, the list box automatically handles standard column sorts when the header is clicked. A standard sort is an alphanumeric sort of column values, alternately ascending/descending with each successive click. All columns are always synchronized automatically.
You can forbid standard user sorts by deselecting the “Sortable” property of the list box.

The developer can set up custom sorts using the **LISTBOX SORT COLUMNS** command and/or combining the On Header Click and On After Sort form events (see the **Form event** command) and array management 4D commands.

**Note**: The “Sortable” column property only affects the standard user sorts; the **LISTBOX SORT COLUMNS** command does not take this property into account.

The value of the variable related to the column header allows you to manage additional information: the current sort of the column (read) and the display of the sort arrow.

- If the variable is set to 0, the column is not sorted and the sort arrow is not displayed;




- If the variable is set to 1, the column is sorted in ascending order and the sort arrow is displayed;




- If the variable is set to 2, the column is sorted in descending order and the sort arrow is displayed.




You can set the value of the variable (for example, **Header2:=2**) in order to “force” the sort arrow display. The column sort itself is not modified in this case; it is up to the developer to handle it.

**Note:** The **OBJECT SET FORMAT** command offers specific support for icons in list box headers, which can be useful when you want to work with a customized sort icon.

## Managing selections

Selections are managed differently depending on whether the list box is based on an array or on a selection.

- **Selection type list box**: Selections are managed by a set, which you can modify if necessary, called *$ListboxSetX* by default (where *X* starts at 0 and is incremented based on the number of list boxes in the form). This set is defined in the properties of the list box. It is automatically maintained by 4D: If the user selects one or more rows in the list box, the set is immediately updated. On the other hand, it is also possible to use the commands of the "Sets" theme in order to modify the selection of the list box via programming.
- **Array type list box**: the **LISTBOX SELECT ROW** command can be used to select one or more rows of the list box by programming.
  The variable linked to the List box object is used to get, set or store selections of object rows. This variable corresponds to a Boolean array that is automatically created and maintained by 4D. The size of this array is determined by the size of the list box: it contains the same number of elements as the smallest array linked to the columns.
  Each element of this array contains **True** if the corresponding line is selected and **False** otherwise. 4D updates the contents of this array depending on user actions. Inversely, you can change the value of array elements to change the selection in the list box.
  On the other hand, you can neither insert nor delete rows in this array; you cannot retype rows either.

**Note**: The **Count in array** command can be used to find out the number of selected lines.

For example, this method allows inverting the selection of the first row of the (array type) list box:

```
ARRAY BOOLEAN(tBListBox;10)
 ` tBListBox is the name of the list box variable in the form
If(tBListBox{1}=True)
   tBListBox{1}:=False
Else
   tBListBox{1}:=True
End if
```

**Note**: The specificities of managing selections in list boxes that are in hierarchical mode are detailed in the **Managing Hierarchical List Boxes** section.

## Printing list boxes

It is possible to print list boxes beginning with 4D v12. Two printing modes are available: preview mode, which can be used to print a list box like a form object, and advanced mode, which lets you control the printing of the list box object itself within the form. Note that the "Printing" appearance is available for list box objects in the Form editor.

### Preview mode

Printing a list box in preview mode consists in directly printing the list box with the form that contains it using the standard print commands or the **Print** menu command. The list box is printed as it is in the form. This mode does not allow precise

control of the printing of the object; in particular, it does not allow you to print all the rows of a list box that contains more rows than it can display.

### Advanced mode

In this mode, the printing of list boxes is carried out by programming, via the **Print object** command (project forms and table forms are supported). The **LISTBOX GET PRINT INFORMATION** command is used to control the printing of the object.

In this mode:

- The height of the list box object is automatically reduced when the number of rows to be printed is less than the original height of the object (there are no "blank" rows printed). On the other hand, the height does not automatically increase according to the contents of the object. The size of the object actually printed can be obtained via the **LISTBOX GET PRINT INFORMATION** command.
- The list box object is printed "as is", in other words, taking its current display parameters into account: visibility of headers and gridlines, hidden and displayed rows, etc.
  These parameters also include the first row to be printed: if you call the **OBJECT SET SCROLL POSITION** command before launching the printing, the first row printed in the list box will be the one designated by the command.
- An automatic mechanism facilitates the printing of list boxes that contain more rows than it is possible to display: successive calls to **Print object** can be used to print a new set of rows each time. The **LISTBOX GET PRINT INFORMATION** command can be used to check the status of the printing while it is underway.

## Managing styles and colors

There are several different ways to set background colors, font colors and font styles for list boxes:

- at the level of the list box object properties,
- at the level of the column properties,
- using arrays or methods for the list box and/or for each column,
- at the level of the text of each cell (if multi-style text).

Priority and inheritance principles are observed.

### Priority

When the same property is set at more than one level, the following priority is applied:

| | |
|---|---|
| *high priority* | Cell (if multi-style text) |
| | Column arrays/methods |
| | List box arrays/methods |
| | Column properties |
| *low priority* | List box properties |

For example, if you set a font style in the list box properties and another using a style array for the column, the latter one will be taken into account.

Given a list box where the rows have an alternating gray/light gray color, defined in the properties of the list box. A background color array has also been set for the list box in order to switch the color of rows where at least one value is negative to light orange:

```
<>_BgndColors{$i}:=0x00FFD0B0 // orange
<>_BgndColors{$i}:=-255 // default value
```

| Header1 | Header2 | Header3 |
|---|---|---|
| 21483 | 9031 | 27290 |
| 24151 | 21990 | -923 |
| 21351 | 2982 | 18009 |
| 8089 | 12898 | 20941 |
| 13001 | -802 | 22059 |
| 4321 | 16826 | 11303 |
| 24082 | 26214 | 22380 |
| 16680 | 23651 | 20403 |
| -2678 | 24818 | 29896 |
| 25639 | 2691 | 9687 |
| 28794 | 26941 | 21486 |
| 26083 | 21092 | 13476 |
| 27928 | 4092 | 15441 |
| 19987 | 28211 | 21191 |
| 7996 | 6300 | 4089 |
| -1063 | 13388 | 23683 |
| 13008 | 7470 | 19897 |
| 5388 | 26918 | 13547 |
| 28559 | 27007 | 8365 |
| 28454 | 22646 | 13824 |

Next you want to color the cells with negative values in dark orange. To do this, you set a background color array for each column, for example *<>_BgndColor_1*, *<>_BgndColor_2* and *<>_BgndColor_3*. The values of these arrays have priority over the ones set in the list box properties as well as those of the general background color array:

```
<>_BgndColorsCol_3{2}:=0x00FF8000 // dark orange
<>_BgndColorsCol_2{5}:=0x00FF8000
<>_BgndColorsCol_1{9}:=0x00FF8000
<>_BgndColorsCol_1{16}:=0x00FF8000
```

| Header1 | Header2 | Header3 |
|---|---|---|
| 21483 | 9031 | 27290 |
| 24151 | 21990 | -923 |
| 21351 | 2982 | 18009 |
| 8089 | 12898 | 20941 |
| 13001 | -802 | 22059 |
| 4321 | 16826 | 11303 |
| 24082 | 26214 | 22380 |
| 16680 | 23651 | 20403 |
| -2678 | 24818 | 29896 |
| 25639 | 2691 | 9687 |
| 28794 | 26941 | 21486 |
| 26083 | 21092 | 13476 |
| 27928 | 4092 | 15441 |
| 19987 | 28211 | 21191 |
| 7996 | 6300 | 4089 |
| -1063 | 13388 | 23683 |
| 13008 | 7470 | 19897 |
| 5388 | 26918 | 13547 |
| 28559 | 27007 | 8365 |
| 28454 | 22646 | 13824 |

You can get the same result using the **LISTBOX SET ROW FONT STYLE** and **LISTBOX SET ROW COLOR** commands. They have the advantage of letting you skip having to predefine style/color arrays for the columns: instead they are created dynamically by the commands.

### Inheritance

For each attribute (style, color and background color), an inheritance is implemented when the default value is used:

- for cell attributes: attribute values of rows
- for row attributes: attribute values of columns
- for column attributes: attribute values of the list box

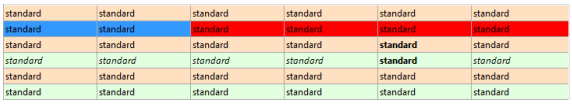
This way, if you want for an object to inherit the attribute value from a higher level, you can use pass the lk inherited constant (default value) to the definition command or directly in the element of the corresponding style/color array.

Given a list box containing a standard font style with alternating colors:

| | | | | | |
|---|---|---|---|---|---|
| standard | standard | standard | standard | standard | standard |
| standard | standard | standard | standard | standard | standard |
| standard | standard | standard | standard | standard | standard |
| standard | standard | standard | standard | standard | standard |
| standard | standard | standard | standard | standard | standard |
| standard | standard | standard | standard | standard | standard |

You perform the following modifications:

- change the background of row 2 to red using the **Row Background Color Array** property of the list box object,
- change the style of row 4 to italics using the **Row Style Array** property of the list box object,
- two elements in column 5 are changed to bold using the **Row Style Array** property of the column 5 object,
- the 2 elements for column 1 and 2 are changed to dark blue using the **Row Background Color Array** property for the column 1 and 2 objects:



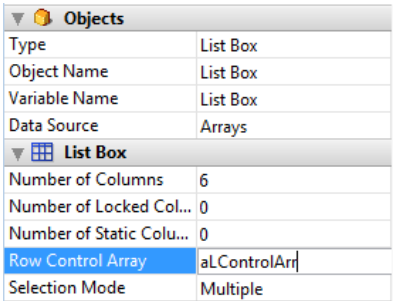
To restore the original appearance of the list box, you can:

- pass the lk inherited constant in element 2 of the background color arrays for columns 1 and 2: then they inherit the red background color of the row.
- pass the lk inherited constant in elements 3 and 4 of the style array for column 5: then they inherit the standard style, except for element 4, which changes to italics as specified in the style array of the list box.
- pass the lk inherited constant in element 4 of the style array for the list box in order to remove the italics style.
- pass the lk inherited constant in element 2 of the background color array for the list box in order to restore the original alternating color of the list box.

## Managing row display

You can set the "hidden", "disabled" and "selectable" interface properties for each row in an array-based list box.

These settings can be handled by means of the **Row Control Array**, which you can designate using the **LISTBOX SET ARRAY** command or through the Property list:



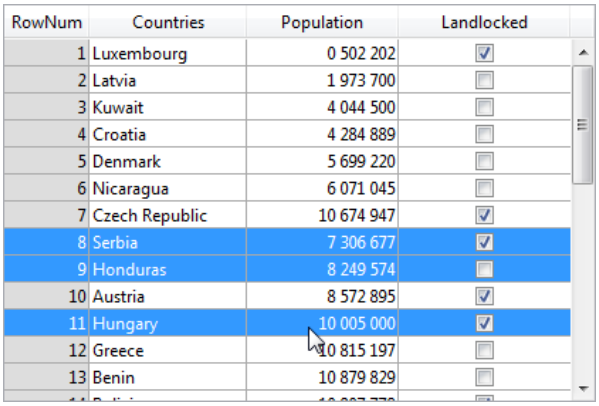

A row control array must be of the Longint type and include the same number of rows as the list box. For more information, refer to the **List box specific properties** section.

Each element of the **Row Control Array** array defines the interface status of its corresponding row in the list box. Three interface properties are available using constants in the "List Box" constant theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk row is disabled | Longint | 2 | The corresponding row is disabled. The text and controls such as check boxes are dimmed or grayed out. Enterable text input areas are no longer enterable. Default value: Enabled |
| lk row is hidden | Longint | 1 | The corresponding row is hidden. Hiding rows only affects the display of the list box. The hidden rows are still present in the arrays and can be managed by programming. The language commands, more particularly **LISTBOX Get number of rows** or **LISTBOX GET CELL POSITION**, do not take the displayed/hidden status of rows into account. For example, in a list box with 10 rows where the first 9 rows are hidden, **LISTBOX Get number of rows** returns 10. From the user's point of view, the presence of hidden rows in a list box is not visibly discernible. Only visible rows can be selected (for example using the Select All command). Default value: Visible |
| lk row is not selectable | Longint | 4 | The corresponding row is not selectable (highlighting is not possible). Enterable text input areas are no longer enterable unless the "Single-Click Edit" option is enabled. Controls such as check boxes and lists are still functional however. This setting is ignored if the list box selection mode is "None". Default value: Selectable |

To change the status for a row, you just need to set the appropriate constant(s) to the corresponding array element. For example, if you do not want row #10 to be selectable, you can write:

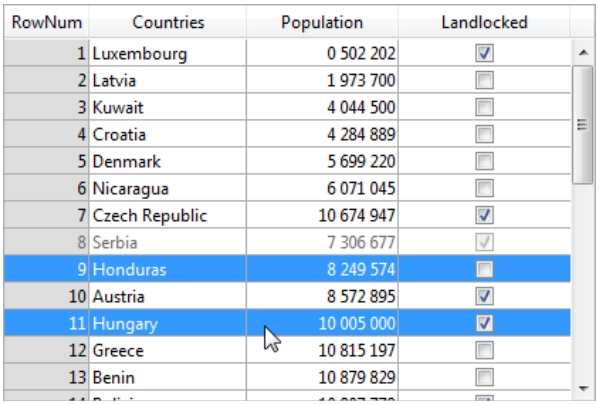
```
aLControlArr{10}:=lk row is not selectable
```



| RowNum | Countries | Population | Landlocked |
|---|---|---|---|
| 1 | Luxembourg | 0 502 202 | ☑ |
| 2 | Latvia | 1 973 700 | ☐ |
| 3 | Kuwait | 4 044 500 | ☐ |
| 4 | Croatia | 4 284 889 | ☐ |
| 5 | Denmark | 5 699 220 | ☐ |
| 6 | Nicaragua | 6 071 045 | ☐ |
| 7 | Czech Republic | 10 674 947 | ☑ |
| 8 | Serbia | 7 306 677 | ☑ |
| 9 | Honduras | 8 249 574 | ☐ |
| 10 | Austria | 8 572 895 | ☑ |
| 11 | Hungary | 10 005 000 | ☑ |
| 12 | Greece | 10 815 197 | ☐ |
| 13 | Benin | 10 879 829 | ☐ |

You can define several interface properties at once:

```
aLControlArr{8}:=lk row is not selectable+lk row is disabled
```



| RowNum | Countries | Population | Landlocked |
|---|---|---|---|
| 1 | Luxembourg | 0 502 202 | ☑ |
| 2 | Latvia | 1 973 700 | ☐ |
| 3 | Kuwait | 4 044 500 | ☐ |
| 4 | Croatia | 4 284 889 | ☐ |
| 5 | Denmark | 5 699 220 | ☐ |
| 6 | Nicaragua | 6 071 045 | ☐ |
| 7 | Czech Republic | 10 674 947 | ☑ |
| 8 | Serbia | 7 306 677 | ☑ |
| 9 | Honduras | 8 249 574 | ☐ |
| 10 | Austria | 8 572 895 | ☑ |
| 11 | Hungary | 10 005 000 | ☑ |
| 12 | Greece | 10 815 197 | ☐ |
| 13 | Benin | 10 879 829 | ☐ |

Note that setting properties for an element overrides any other values for this element (if not reset). For example:

```
aLControlArr{6}:=lk row is disabled+lk row is not selectable //sets row 6 as disabled AND not selectable
aLControlArr{6}:=lk row is disabled //sets row 6 as disabled but selectable again
```

## Displaying the result of an SQL query in a list box

It is possible to place the results of an SQL query directly in an array type list box. This offers a rapid means for viewing the results of SQL queries. Only queries of the **SELECT** type can be used. This mechanism cannot be used with an external SQL database.

It works according to the following principles:

- Create the list box which will receive the query results. The data source of the list box must be **Arrays**.
- Execute an SQL query of the **SELECT** type and assign the result to the variable associated with the list box. You can use the **Begin SQL**/**End SQL** keywords (see the 4D *Language Reference* manual).
- List box columns can be sorted or modified by the user.
- Each new execution of a **SELECT** query with the list box leads to the resetting of the columns (it is not possible to fill the same list box progressively using several **SELECT** queries).
- It is recommended to give the list box the same number of columns as there will be in the SQL query result. If the number of list box columns is less than that required by the **SELECT** query, columns are added automatically. If the number of columns is more than required by the **SELECT** query, the unnecessary columns are automatically hidden. **Note:** The columns added automatically are bound to **Dynamic variables** of the array type. These dynamic arrays last as long as the form does. A dynamic variable is also created for each header. When the **LISTBOX GET ARRAYS** command is called, the *arrColVars* parameter contains pointers to the dynamic arrays and the *arrHeaderVars* parameter contains pointers to the dynamic header variables. If the added column is, for example, the fifth column, its name is *sql_column5* and its header name is *sql_header5*.
- In interpreted mode, existing arrays that are used by the list box can be retyped automatically according to the data sent by the SQL query.

**Example**

We want to retrieve all the fields of the PEOPLE table and put their contents into the list box having the variable name *vlistbox*. In the object method of a button (for example), simply write:

```
Begin SQL
   SELECT * FROM PEOPLE INTO <<vlistbox>>
End SQL
```

# Managing Hierarchical List Boxes

4D lets you specify and use hierarchical list boxes. A hierarchical list box is a list box in which the contents of the first column appears in hierarchical form. This type of representation is adapted to the presentation of information including repeated values and/or values that are hierarchically dependent (country/region/city and so on).

Only **list boxes of the array type** can be hierarchical.

Hierarchical list boxes are a particular way of representing data but they do not modify the structure of these data (the arrays). Hierarchical list boxes are filled and managed exactly the same way as regular list boxes (see **Managing List Box Objects**).

To specify a hierarchical list box, there are three different possibilities:

- manually configure hierarchical elements using the Property list of the form editor or using the list box management pop-up menu. These points are covered in the *Design Reference* manual of 4D.
- use the **LISTBOX SET HIERARCHY** and **LISTBOX GET HIERARCHY** commands.

When a form containing a hierarchical list box is opened for the first time, by default all the rows are expanded.
A break row and a hierarchical "node" are automatically added in the list box when values are repeated in the arrays. For example, imagine a list box containing four arrays specifying cities, each city being characterized by a country, a region, a name and a number of inhabitants:

| Country | Region | City | Population |
|---|---|---|---|
| France | Brittany | Rennes | 200000 |
| France | Brittany | Quimper | 80000 |
| France | Brittany | Brest | 120000 |
| France | Normandy | Caen | 75000 |
| France | Normandy | Deauville | 35000 |

If this list box is displayed in hierarchical form (the first three arrays being included in the hierarchy), you will obtain:

| City | Population |
|---|---|
| France | |
| Brittany | |
| Rennes | 200000 |
| Quimper | 80000 |
| Brest | 120000 |
| Normandy | |
| Caen | 75000 |
| Deauville | 35000 |

The arrays are not sorted before the hierarchy is constructed. If, for example, an array contains the data AAABBAACC, the hierarchy obtained will be:

> A
> B
> A
> C

To expand or collapse a hierarchical "node", simply click on it. If you **Alt+click** (Windows) or **Option+click** (Mac OS) on the node, all its sub-elements will be expanded or collapsed. These operations can also be carried out by programming using the **LISTBOX EXPAND** and **LISTBOX COLLAPSE** commands.

## Management of selections and positions

A hierarchical list box displays a variable number of rows on screen according to the expanded/collapsed state of the hierarchical nodes. This does not however mean that the number of rows of the arrays vary. Only the display is modified, not the data.

It is important to understand this principle because programmed management of hierarchical list boxes is always based on the data of the arrays, not on the displayed data. In particular, the break rows added automatically are not taken into account in the display options arrays (see below).

Let’s look at the following arrays for example:

| | | |
|---|---|---|
| France | Brittany | Brest |
| France | Brittany | Quimper |
| France | Brittany | Rennes |

If these arrays are represented hierarchically, the row "Quimper" will not be displayed on the second row, but on the fourth, because of the two break rows that are added:

| |
|---|
| France |
| Brittany |
| Brest |
| Quimper |
| Rennes |

Regardless of how the data are displayed in the list box (hierarchically or not), if you want to change the row containing "Quimper" to bold, you must use the statement Style{2} = bold. Only the position of the row in the arrays is taken into account.

This principle is implemented for internal arrays that can be used to manage:

- colors
- background colors
- styles
- hidden rows
- selections

For example, if you want to select the row containing Rennes, you must pass:

```
->MyListbox{3}:=True
```

*Non-hierarchical representation:*

| | | |
|---|---|---|
| France | Brittany | Brest |
| France | Brittany | Quimper |
| France | Brittany | Rennes |

*Hierarchical representation:*

| |
|---|
| France |
| Brittany |
| Brest |
| Quimper |
| Rennes |

**Note:** If one or more rows are hidden because their parents are collapsed, they are no longer selected. Only the rows that are visible (either directly or by scrolling) can be selected. In other words, rows cannot be both hidden and selected.

As with selections, the **LISTBOX GET CELL POSITION** command will return the same values for a hierarchical list box and a non-hierarchical list box. This means that in both of the examples below, **LISTBOX GET CELL POSITION** will return the same position: (3;2).

*Non-hierarchical representation:*

| | | | |
|---|---|---|---|
| France | Brittany | Brest | 120000 |
| France | Brittany | Quimper | 80000 |
| France | Brittany | Rennes | 200000 |
| France | Normandy | Caen | 75000 |

*Hierarchical representation:*

| | |
|---|---|
| France | |
| Brittany | |
| Brest | 120000 |
| Quimper | 80000 |
| Rennes | 200000 |
| Normandy | |
| Caen | 75000 |

## Management of break rows

If the user selects a break row, **LISTBOX GET CELL POSITION** returns the first occurrence of the row in the corresponding array. In the following case:

| | |
|---|---|
| France | |
| Brittany | |
| Brest | 120000 |
| Quimper | 80000 |
| Rennes | 200000 |
| Normandy | |
| Caen | 75000 |

... **LISTBOX GET CELL POSITION** returns (2;4). To select a break row by programming, you will need to use the **LISTBOX SELECT BREAK** command.

Break rows are not taken into account in the internal arrays used to manage the graphic appearance of list boxes (styles and colors). It is however possible to modify these characteristics for break rows via the graphic management commands for objects (**Objects (Forms)** theme). You simply need to execute the appropriate commands on the arrays that constitute the hierarchy.

Given for example the following list box (the names of the associated arrays are specified in parentheses):

*Non-hierarchical representation:*

| (T1) | (T2) | (T3) | (T4) | (tStyle) | (tColor) |
|---|---|---|---|---|---|
| France | Brittany | Brest | 120000 | *Normal* | *0* |
| France | Brittany | Quimper | 80000 | *Underline* | *0* |
| France | Brittany | Rennes | 200000 | *Normal* | *0xFF0000* |
| France | Normandy | Caen | 220000 | *Normal* | *0* |
| France | Normandy | Deauville | 4000 | *Normal* | *0* |

*Hierarchical representation:*

| | |
|---|---|
| France | |
| Brittany | |
| Brest | 120000 |
| Quimper | 80000 |
| Rennes | 200000 |
| Normandy | |
| Caen | 75000 |
| Deauville | 4000 |

In hierarchical mode, break levels are not taken into account by the style modification arrays named *tStyle* and *tColors*. To modify the color or style of break levels, you must execute the following statements:

```
OBJECT SET RGB COLORS(T1;0x0000FF;0xB0B0B0)
OBJECT SET FONT STYLE(T2;Bold)
```

**Note:** In this context, only the syntax using the array variable can function with the object property commands because the arrays do not have any associated object.

Result:

| | |
|---|---|
| France | |
| **Brittany** | |
| Brest | 120000 |
| Quimper | 80000 |
| Rennes | 200000 |
| **Normandy** | |
| Caen | 75000 |
| Deauville | 4000 |

### Hidden rows

When all the rows of a sub-hierarchy are hidden, the break line is automatically hidden. In the above example, if rows 1 to 3 are hidden, the "Brittany" break row will not appear.

## Optimized management of expand/collapse

You can optimize hierarchical list boxes display and management using the On Expand and On Collapse form events.

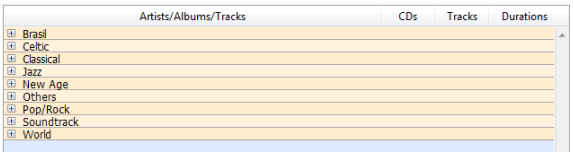
A hierarchical list box is built from the contents of its arrays so it can only be displayed when all these arrays are loaded into memory. This makes it difficult to build large hierarchical list boxes based on arrays generated from data (through the **SELECTION TO ARRAY** command), not only because of the display speed but also the memory used.

Using the On Expand and On Collapse form events can overcome these constraints: for example, you can now display only part of the hierarchy and load/unload the arrays on the fly, based on user actions.

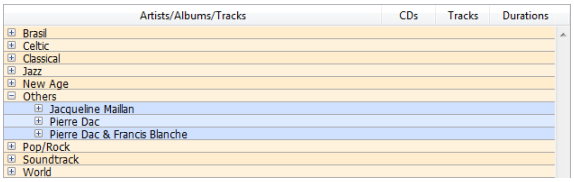
In the context of these events, the **LISTBOX GET CELL POSITION** command returns the cell where the user clicked in order to expand or collapse a row.

In this case, you must fill and empty arrays through the code. The principles to be implemented are:

- When the list box is displayed, only the first array must be filled. However, you must create a second array with empty values so that the list box displays the expand/collapse buttons:




- When a user clicks on an expand button, you can process the On Expand event. The **LISTBOX GET CELL POSITION** command returns the cell concerned and lets you build the appropriate hierarchy: you fill the first array with the repeated values and the second with the values sent from the **SELECTION TO ARRAY** command and you insert as many rows as needed in the list box using the **LISTBOX INSERT ROWS** command.

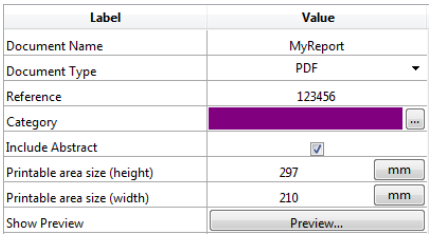



- When a user clicks on a collapse button, you can process the On Collapse event. The **LISTBOX GET CELL POSITION** command returns the cell concerned: you remove as many rows as needed from the list box using the **LISTBOX DELETE ROWS** command.

# Using object arrays in columns (4D View Pro)

List box columns can work with object arrays. Since object arrays can contain different kinds of data, this powerful feature allows you to enter and display various types of values in the rows of a single column, and use various widgets as well. For example, you could insert a text input in the first row, a check box in the second, and a drop-down list in the third. Object arrays also provide access to new kinds of widgets, such as buttons or color pickers.

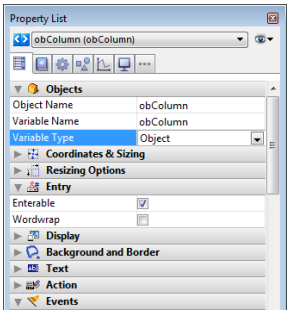
The following list box was designed using an object array:




**About 4D View Pro:** The ability to use object arrays in list boxes is a "4D View Pro" function, which means that its use requires you to have a valid 4D View license. 4D View Pro is a new tool being developed which includes a set of features related to arrays and list presentations. It is intended to progressively replace the 4D View plug-in. For more information, please refer to the 4D Web site.

## Configuring an object array column

To assign an object array to a list box column, you just need to set the object array name in either the Property list ("Variable Name" field), or using the **LISTBOX INSERT COLUMN** command, like with any array-based column. In the Property list, you can select **Object** as a "Variable Type" for the column:



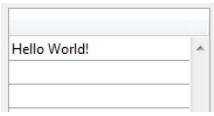

Standard properties related to coordinates, size, and style are available for object columns. You can define them using the Property list, or by programming the style, font color, background color and visibility for each row of an object-type list box column. These types of columns can also be hidden.

However, the **Data Source** theme is not available for object-type list box columns. In fact, the contents of each column cell are based on attributes found in the corresponding element of the object array. Each array element can define:

- the value type (*mandatory*): text, color, event, etc.
- the value itself (*optional*): used for input/output.
- the cell content display (*optional*): button, list, etc.
- additional settings (*optional*): depend on the value type

To define these properties, you need to set the appropriate attributes in the object (available attributes are listed below). For example, you can write "Hello World!" in an object column using this simple code:

```
ARRAY OBJECT(obColumn;0) //column array
C_OBJECT($ob) //first element
OB SET($ob;"valueType";"text") //defines the value type (mandatory)
OB SET($ob;"value";"Hello World!") //defines the value
APPEND TO ARRAY(obColumn;$ob)
```




**Note:** Display format and entry filters cannot be set for an object column. They automatically depend on the value type.

## valueType and data display

When a list box column is associated with an object array, the way a cell is displayed, entered, or edited, is based on the **valueType** attribute of the array element. Supported **valueType** values are:

- "text": for a text value
- "real": for a numeric value that can include separators like a <space>, <.>, or <,>
- "integer": for an integer value
- "boolean": for a True/False value
- "color": to define a background color
- "event": to display a button with a label.

4D uses default widgets with regards to the "valueType" value (i.e., a "text" is displayed as a text input widget, a "boolean" as a check box), but alternate displays are also available through options (e.g., a real can also be represented as a drop-down menu). The following table shows the default display as well as alternatives for each type of value:

| valueType | Default widget | Alternative widget(s) |
|---|---|---|
| text | text input | drop-down menu (required list) or combo box (choice list) |
| real | controlled text input (numbers and separators) | drop-down menu (required list) or combo box (choice list) |
| integer | controlled text input (numbers only) | drop-down menu (required list) or combo box (choice list) or three-states check box |
| boolean | check box | drop-down menu (required list) |
| color | background color | text |
| event | button with label | |
| | | All widgets can have an additional **unit toggle button** or **ellipsis button** attached to the cell. |

You set the cell display and options using specific attributes in each object (see below).

### Display formats and entry filters

You cannot set display formats or entry filters for columns of object-type list boxes. They are automatically defined according to the value type. These are listed in the following table:

| Value type | Default format | Entry control |
|---|---|---|
| text | same as defined in object | any (no control) |
| real | same as defined in object (using system decimal separator) | "0-9" and "." and "-" |
| | | "0-9" and "." if min>=0 |
| integer | same as defined in object | "0-9" and "-" |
| | | "0-9" if min>=0 |
| Boolean | check box | N/A |
| color | N/A | N/A |
| event | N/A | N/A |

## Attributes

Each element of the object array is an object that can contain one or more attributes that will define the cell contents and data display (see example above).

The only mandatory attribute is "valueType" and its supported values are "text", "real", "integer", "boolean", "color", and "event". The following table lists all the attributes supported in list box object arrays, depending on the "valueType" value (any other attributes are ignored). Display formats are detailed and examples are provided below.

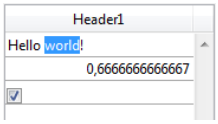
| | valueType | text | real | integer | boolean | color | event |
|---|---|---|---|---|---|---|---|
| ***Attributes*** | ***Description*** | | | | | | |
| **value** | cell value (input or output) | x | x | x | | | |
| **min** | minimum value | | x | x | | | |
| **max** | maximum value | | x | x | | | |
| **behavior** | "threeStates" value | | | x | | | |
| **requiredList** | drop-down list defined in object | x | x | x | | | |
| **choiceList** | combo box defined in object | x | x | x | | | |
| **requiredListReference** | 4D list ref, depends on "saveAs" value | x | x | x | | | |
| **requiredListName** | 4D list name, depends on "saveAs" value | x | x | x | | | |
| **saveAs** | "reference" or "value" | x | x | x | | | |
| **choiceListReference** | 4D list ref, display combo box | x | x | x | | | |
| **choiceListName** | 4D list name, display combo box | x | x | x | | | |
| **unitList** | array of X elements | x | x | x | | | |
| **unitReference** | index of selected element | x | x | x | | | |
| **unitsListReference** | 4D list ref for units | x | x | x | | | |
| **unitsListName** | 4D list name for units | x | x | x | | | |
| **alternateButton** | add an alternate button | x | x | x | x | x | |

### value

Cell values are stored in the "value" attribute. This attribute is used for input as well as output. It can also be used to define default values when using lists (see below).

**Example:**

```
ARRAY OBJECT(obColumn;0) //column array
C_OBJECT($ob1)
$entry:="Hello world!"
OB SET($ob1;"valueType";"text")
OB SET($ob1;"value";$entry) // if the user enters a new value, $entry will contain the edited value
C_OBJECT($ob2)
OB SET($ob2;"valueType";"real")
OB SET($ob2;"value";2/3)
C_OBJECT($ob3)
OB SET($ob3;"valueType";"boolean")
OB SET($ob3;"value";True)

APPEND TO ARRAY(obColumn;$ob1)
APPEND TO ARRAY(obColumn;$ob2)
APPEND TO ARRAY(obColumn;$ob3)
```




**Note:** Null values are supported and result in an empty cell.

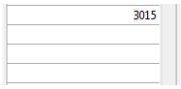
### min and max

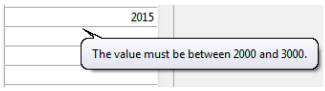
When the "valueType" is "real" or "integer", the object also accepts **min** and **max** attributes with appropriate values (values must be of the same type as the valueType).

These attributes can be used to control the range of input values. When a cell is validated (when it loses the focus), if the input value is lower than the **min** value or greater than the **max** value, then it is rejected. In this case, the previous value is maintained and a tip displays an explanation.

**Example:**

```
C_OBJECT($ob3)
$entry3:=2015
OB SET($ob3;"valueType";"integer")
OB SET($ob3;"value";$entry3)
OB SET($ob3;"min";2000)
OB SET($ob3;"max";3000)
```




### behavior

The **behavior** attribute provides variations to the regular representation of values. In the current version of 4D, a single variation is proposed:

| Attribute | Available value(s) | valueType(s) | Description |
|---|---|---|---|
| behavior | threeStates | integer | Represents a numeric value as a three-states check box. 2=semi-checked, 1=checked, 0=unchecked, -1=invisible, -2=unchecked disabled, -3=checked disabled, -4=semi-checked disabled |

**Example:**

```
C_OBJECT($ob3)
OB SET($ob3;"valueType";"integer")
OB SET($ob3;"value";-3)
C_OBJECT($ob4)
OB SET($ob4;"valueType";"integer")
OB SET($ob4;"value";-3)
OB SET($ob4;"behavior";"threeStates")
```


### requiredList and choiceList

When a "choiceList" or a "requiredList" attribute is present inside the object, the text input is replaced by a drop-down list or a combo box, depending of the attribute:

- If the attribute is "choiceList", the cell is displayed as a combo box. This means that the user can select or type a value.
- If the attribute is "requiredList" then the cell is displayed as a drop-down list and the user can only select one of the values provided in the list.

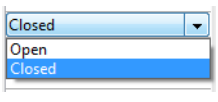
In both cases, a "value" attribute can be used to preselect a value in the widget.

**Note:** The widget values are defined through an array. If you want to assign an existing 4D list to the widget, you need to use the "requiredListReference", "requiredListName", "choiceListReference", or "choiceListName" attributes.

**Examples:**

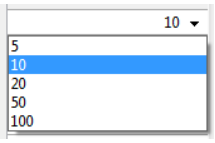
- You want to display a drop-down list with only two options: "Open" or "Closed". "Closed" must be preselected:

```
ARRAY TEXT($RequiredList;0)
APPEND TO ARRAY($RequiredList;"Open")
APPEND TO ARRAY($RequiredList;"Closed")
C_OBJECT($ob)
OB SET($ob;"valueType";"text")
OB SET($ob;"value";"Closed")
OB SET ARRAY($ob;"requiredList";$RequiredList)
```




- You want to accept any integer value, but display a combo box to suggest the most common values:

```
ARRAY LONGINT($ChoiceList;0)
APPEND TO ARRAY($ChoiceList;5)
APPEND TO ARRAY($ChoiceList;10)
APPEND TO ARRAY($ChoiceList;20)
APPEND TO ARRAY($ChoiceList;50)
APPEND TO ARRAY($ChoiceList;100)
C_OBJECT($ob)
OB SET($ob;"valueType";"integer")
OB SET($ob;"value";10) //10 as default value
OB SET ARRAY($ob;"choiceList";$ChoiceList)
```




### requiredListName and requiredListReference

The "requiredListName" and "requiredListReference" attributes allow you to use, in a list box cell, a list defined in 4D either in Design mode (in the **Lists** editor of the Tool box) or by programming (using the **New list** command). The cell will then be displayed as a drop-down list. This means that the user can only select one of the values provided in the list.

Use "requiredListName" or "requiredListReference" depending on the origin of the list: if the list comes from the Tool box, you pass a name; otherwise, if the list has been defined by programming, you pass a reference. In both cases, a "value" attribute can be used to preselect a value in the widget.

**Note:** If you want to define these values through a simple array, you need to use the "requiredList" attribute.

In this case, the "saveAs" attribute will define whether the selected item must be saved as a "value" or as a "reference".

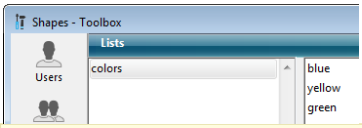
- If "saveAs" = "reference" then it will be saved as a reference and the "valueType" must be real or integer.
- If "saveAs" = "value" then the value is saved. In this case, the "valueType" must be the same type as the values of the list, usually "text" or "integer"; otherwise 4D will try to convert the value of the list into the "valueType" of the object (see examples below).

For more information about the "save as" option, please refer to the **Save as Value or Reference** section in the *Design Reference* manual.

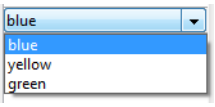
**Note:** If the list contains text items representing real values, the decimal separator must be a period ("."), regardless of the local settings, e.g.: "17.6" "1234.456".

**Examples:**

- You want to display a drop-down list based on a "colors" list defined in the Tool box (containing the values "blue", "yellow", and "green"), save it as a value and display "blue" by default:

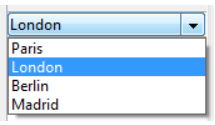



```
C_OBJECT($ob)
OB SET($ob;"valueType";"text")
OB SET($ob;"saveAs";"value")
OB SET($ob;"value";"blue")
OB SET($ob;"requiredListName";"colors")
```




- You want to display a drop-down list based on a list defined by programming and save it as a reference:

```
<>List:=New list
APPEND TO LIST(<>List;"Paris";1)
APPEND TO LIST(<>List;"London";2)
APPEND TO LIST(<>List;"Berlin";3)
APPEND TO LIST(<>List;"Madrid";4)
C_OBJECT($ob)
OB SET($ob;"valueType";"integer")
OB SET($ob;"saveAs";"reference")
OB SET($ob;"value";2) //displays London by default
OB SET($ob;"requiredListReference";<>List)
```




### choiceListName and choiceListReference

The "choiceListName" and "choiceListReference" attributes allow you to use, in a list box cell, a list defined in 4D either in Design mode (in the Tool box) or by programming (using the **New list** command). The cell is then displayed as a combo box, which means that the user can select or type a value.

Use "choiceListName" or "choiceListReference" depending on the origin of the list: if the list comes from the Tool box, you pass a name; otherwise, if the list has been defined by programming, you pass a reference. In both cases, a "value" attribute can be used to preselect a value in the widget.

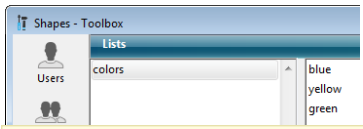
**Note:** If you want to define these values through a simple array, you need to use the "choiceList" attribute.

The "saveAs" attribute cannot be used in this case because selected items are automatically saved as a "value" (cf. for more information).

**Note:** If the list contains text items representing real values, the decimal separator must be a period ("."), regardless of the local settings, e.g.: "17.6" "1234.456".

**Example:**

You want to display a combo box based on a "colors" list defined in the Tool box (containing the values "blue", "yellow", and "green") and display "green" by default:




```
C_OBJECT($ob)
OB SET($ob;"valueType";"text")
OB SET($ob;"value";"blue")
OB SET($ob;"choiceListName";"colors")
```


### unitsList, unitsListName, unitsListReference and unitReference

You can use specific attributes to add units associated with cell values (e.g.: "10 cm", "20 pixels", etc.). To define the unit list, you can use one of the following attributes:

- "unitsList": an array containing the *x* elements used to define the available units (e.g.: "cm", "inches", "km", "miles", etc.). Use this attribute to define units within the object.
- "unitsListReference": a reference to a 4D list containing available units. Use this attribute to define units with a 4D list created with the **New list** command.
- "unitsListName": a name of a design-based 4D list that contains available units. Use this attribute to define units with a 4D list created in the Tool box.

Regardless of the way the unit list is defined, it can be associated with the following attribute:

- "unitReference": a single value that contains the index (from 1 to *x*) of the selected item in the "unitList", "unitsListReference" or "unitsListName" values list.

The current unit is displayed as a button that cycles through the "unitList", "unitsListReference" or "unitsListName" values each time it is clicked (e.g., "pixels" -> "rows" -> "cm" -> "pixels" -> etc.)

**Example:** We want to set up a numeric input followed by two possible units: "rows" or "pixels". The current value is "2" + "lines". We use values defined directly in the object ("unitsList" attribute):

```
ARRAY TEXT($_units;0)
APPEND TO ARRAY($_units;"lines")
APPEND TO ARRAY($_units;"pixels")
C_OBJECT($ob)
OB SET($ob;"valueType";"integer")
OB SET($ob;"value";2) // 2 "units"
OB SET($ob;"unitReference";1) //"lines"
OB SET ARRAY($ob;"unitsList";$_units)
```

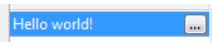

### alternateButton

If you want to add an ellipsis button [**...**] to a cell, you just need to pass the "alternateButton" with the **True** value in the object. The button will be displayed in the cell automatically.

When this button is clicked by a user, an On Alternate Click event will be generated, and you will be able to handle it however you want (see the "**Event management**" section below for more information).

**Example:**

```
C_OBJECT($ob1)
$entry:="Hello world!"
OB SET($ob;"valueType";"text")
OB SET($ob;"alternateButton";True)
OB SET($ob;"value";$entry)
```




## color valueType

The "color" *valueType* allows you to display either a color or a text.

- If the value is a number, a colored rectangle is drawn inside the cell. Example:

```
C_OBJECT($ob4)
OB SET($ob4;"valueType";"color")
OB SET($ob4;"value";0x00FF0000)
```

- If the value is a text, then the text is displayed (e.g.: "value";"Automatic").

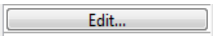
## event valueType

The "event" *valueType* displays a simple button that generates an On Clicked event when clicked. No data or value can be passed or returned.

Optionally, you can pass a "label" attribute.

Example:

```
C_OBJECT($ob)
OB SET($ob;"valueType";"event")
OB SET($ob;"label";"Edit...")
```




## Event management

Several events can be handled while using an object list box array:

- **On Data Change**: An On Data Change event is triggered when any value has been modified either:
  - in a text input zone
  - in a drop-down list
  - in a combo box area
  - in a unit button (switch from value x to value x+1)
  - in a check box (switch between checked/unchecked)
- **On Clicked**: When the user clicks on a button installed using the "event" *valueType* attribute, an On Clicked event will be generated. This event is managed by the programmer.
- **On Alternative Click**: When the user clicks on an ellipsis button ("alternateButton" attribute), an On Alternative Click event will be generated. This event is managed by the programmer.

**Compatibility note (4D v15): On Alternative Click** is the new name of the **On Arrow Click** event that was available in previous versions of 4D. This event has been renamed since its scope has been extended.

# 4D View Pro

## About 4D View Pro

4D View Pro is a new tool being developed by 4D. It provides a set of advanced features related to arrays and list presentations, and is essentially based upon list box objects. 4D View Pro offers 4D users a modern and integrated alternative to some of the legacy 4D View product features.

## 4D View Pro feature list

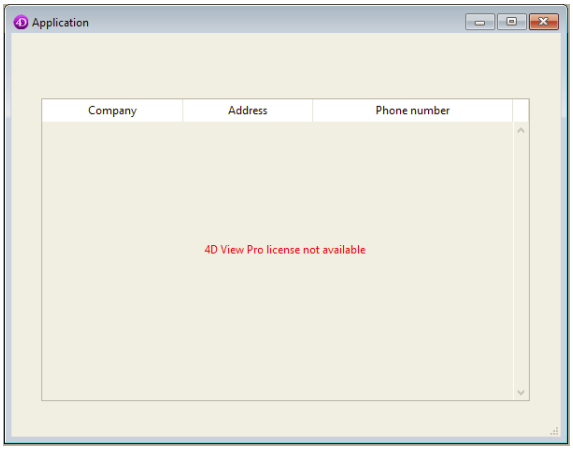
In the current version of the program, 4D View Pro provides the following features:

- Object arrays associated with a list box column (see **Using object arrays in columns (4D View Pro)**)
- Variable row height within a list box:
  - **Row Height Array** property
  - **LISTBOX Get row height** and **LISTBOX SET ROW HEIGHT** commands
  - lk row height array constant for **LISTBOX Get array** and **LISTBOX SET ARRAY** commands

## Installation and activation

Unlike with the legacy 4D View product, 4D View Pro features are directly included in 4D itself, making it easier to deploy and manage. No additional installation is required.

However, 4D View Pro requires the same license as 4D View. You need to have this license installed in your application in order to enable these features. When the 4D View license is not installed, the contents of a list box that uses a 4D View Pro feature are not displayed at runtime and an error message is displayed instead:



# LISTBOX COLLAPSE

LISTBOX COLLAPSE ( {* ;} object {; recursive {; selector {; line {; column}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| recursive | Boolean | ⇒ | True = collapse sublevels False = do not collapse sublevels |
| selector | Longint | ⇒ | Part of list box to collapse |
| line | Longint | ⇒ | Number of break row to collapse or<br>Number of list box level to collapse |
| column | Longint | ⇒ | Number of break column to collapse |

## Description

The **LISTBOX COLLAPSE** command is used to collapse the break rows of the list box object designated by the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string.

If the list box is not configured in hierarchical mode, the command does nothing. For more information about hierarchical list boxes, please refer to the **Managing Hierarchical List Boxes** section.

The optional *recursive* parameter is used to configure the collapsing of the hierarchical sublevels of the list box. Pass True or omit this parameter for the command to collapse all the levels and all the sublevels. If you pass False, only the first level will be collapsed.

The optional *selector* parameter is used to specify the scope of the command. You can pass one of the following constants, found in the **List Box** theme, in this parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk all | Longint | 0 | The command affects all sub-levels (default value, used when parameter is omitted). |
| lk selection | Longint | 1 | The command affects selected sub-levels. |
| lk break row | Longint | 2 | The command affects the sub-level to which the "cell" designated by the *row* and *column* parameters belongs. Note that these parameters represent the row and column numbers in the list box in standard mode and not in its hierarchical representation. If the *row* and *column* parameters are omitted, the command does nothing. |
| lk level | Longint | 3 | The command affects all the break rows corresponding to the *level* column. This parameter designates a column number in the list box in standard mode and not in its hierarchical representation. If the *level* parameter is omitted, the command does nothing. |

If the selection or list box does not contain a break row or if all the break rows are already collapsed, the command does nothing.

## Example

This example collapses the first level of the break rows of the selection in the list box:

```
LISTBOX COLLAPSE(*;"MyListbox";False;lk selection)
```

# LISTBOX DELETE COLUMN

LISTBOX DELETE COLUMN ( {* ;} object ; colPosition {; number} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or Variable (if * is omitted) |
| colPosition | Longint | ➡ | Column number to remove |
| number | Longint | ➡ | Number of columns to be removed |

## Description

The **LISTBOX DELETE COLUMN** command removes one or more columns (visible or invisible) in the list box set in the *object* and * parameters.

**Note:** This command does nothing if it is applied to the first column of a list box displayed in hierarchical mode.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

If you do not pass the optional *number* parameter, the command simply removes the column set in the *colPosition* parameter.

Otherwise, the *number* parameter indicates the number of columns to remove to the right starting from the column *colPosition* (this one included).

If the *colPosition* parameter is greater than the number of columns in the list box, the command does nothing.

# LISTBOX DELETE ROWS

LISTBOX DELETE ROWS ( {* ;} object ; rowPosition {; numRows} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| rowPosition | Longint | ⇒ | Position of the row to delete |
| numRows | Longint | ⇒ | Number of rows to delete |

## Description

The **LISTBOX DELETE ROWS** command deletes one or several row(s) starting at *rowPosition* row (visible or not) from the list box set in the *object* and * parameters.

**Note:** This command only works with list boxes based on arrays. When this command is used with a list box based on a selection, it does nothing and the OK system variable is set to 0.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

Keep in mind that after command execution, there will no longer be any element selected in the list box.

The *rowPosition* row is automatically removed from all the arrays used by the list box columns.

If the *rowPosition* value is higher than the total number of rows in the list box, or if it is less than 1, the command does nothing.

**Note:** This command does not take into account any hidden/displayed states of list box rows.

# LISTBOX DUPLICATE COLUMN

LISTBOX DUPLICATE COLUMN ( {* ;} object ; colPosition ; colName ; colVariable ; headerName ; headerVar {; footerName ; footerVar} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or **Variable (if * is omitted) of the column to be duplicated** |
| colPosition | Longint | ⇒ | Location of new duplicated column |
| colName | String | ⇒ | Name of new column |
| colVariable | Array, Field, Variable, Nil pointer | ⇒ | Name of the column array variable or field or variable |
| headerName | String | ⇒ | Column header object name |
| headerVar | Integer variable, Nil pointer | ⇒ | Column header variable |
| footerName | String | ⇒ | Column footer object name |
| footerVar | Variable, Nil pointer | ⇒ | Column footer variable |

## Description

The **LISTBOX DUPLICATE COLUMN** command duplicates the column designated by the *object* and * parameters by programming in the context of the form being executed (Application mode). The original form, generated in the Design mode, is not modified.

**Note:** This functionality was already found in 4D, in Design mode only, using the **Duplicate Column** command found in the context menu of the Form editor.

By default, all style options (size, color, formats, etc.) set for the source column by means of the Property list or using object management commands (**OBJECT SET COLOR**, etc.) are applied to the copy. The object method and the settings of the form events are also duplicated.
However, the data source (array or selection, depending on the source type set for the list box) as well as the style and color arrays are not duplicated. It is your responsibility to define them for each new column after duplication.

The *object* and * parameters designate the column to duplicate. Passing the optional * parameter indicates that the object parameter is an column name (string). If you do not pass this parameter, it indicates that the *object* parameter is a column variable. In this case, you pass a variable reference instead of a string.

**Note:** This command does nothing when it is applied to the first column of a list box displayed in hierarchical mode.

The new duplicated column is placed just before the column designated by the *colPosition* parameter. If the *colPosition* parameter is greater than the total number of columns, then the duplicated column is placed after the last column.

In the *colName* and *colVariable* parameters, pass the object name and the variable of the new duplicated column.

- For array type list boxes, the variable name corresponds to the name of the array whose contents will be displayed in the column. You can pass a Nil (->[]) pointer in a dynamic context (see below).
- For selection type list boxes, you can pass a field or variable in the *colVariable* parameter. So the contents of the column will be the value of this field or variable, evaluated for each record of the selection associated with the list box. This type of content can only be used when the "Data Source" property of the list box is set to Current Selection or Named Selection.

Remember that the data source of the original column is not duplicated: you must set a source variable, array or field for the new duplicated column.

In the *headerName* and *headerVariable* parameters, pass the object name and variable for the header of the new duplicated column. You can also pass the object name and variable for the footer of the inserted column in the *footerName* and *footerVariable* parameters. If you omit the *footerVariable* parameter, 4D uses a dynamic variable.

**Note:** Object names must be unique in a form. You must make sure that the names passed in the *colName*, *headerName* and *footerName* parameters have not already been used. Otherwise, the column is not duplicated and an error is generated.

This command must be used in the context of displaying a form. It is usually called in the On Load form event or following a user action (On Clicked event).

### Dynamic duplication

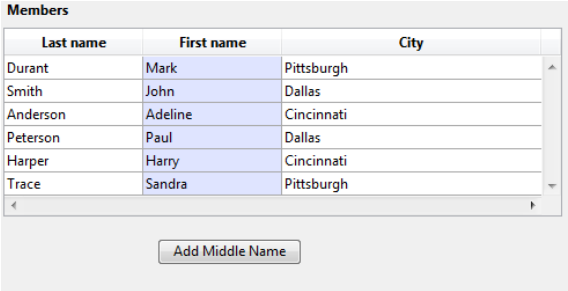
Starting with 4D v14 R3, you can duplicate list box columns dynamically and 4D will automatically handle the definition of the necessary variables (column, footer and header).

To do this, **LISTBOX DUPLICATE COLUMN** accepts a **Nil** (**->[]**) pointer as a value for the *colVariable* (array type list box only), *headerVar* and *footerVar* parameters. In this case, when the command is executed, 4D creates the required variables dynamically (for more information, refer to the **Dynamic variables** section).

Note that header and footer variables are always created with a specific type (longint and text, respectively). Conversely, column variables cannot be typed when created because list boxes accept different types of arrays for these variables (text array, integer array, and so on). This means you have to set the array type manually (see example 2). It is important to perform this typing before calling commands such as **LISTBOX INSERT ROWS** to insert new elements in the array. Alternatively, you can use **APPEND TO ARRAY** both for setting the type of the array and inserting elements.

## Example 1

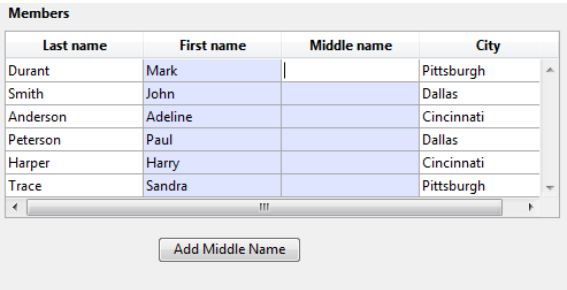
In an array type list box, we want to duplicate the "First Name" column, ready for input:




Here is the code of the button:

```
ARRAY TEXT(arrFirstNames2;Records in table([Members]))
LISTBOX DUPLICATE COLUMN(*;"column2";3;"col2bis";arrFirstNames2;"FirstNameA";vHead2A)
OBJECT SET TITLE(*;"FirstNameA";"Middle Name")
EDIT ITEM(*;"col2A";0)
```

When you click on the button, the list box appears as follows:




## Example 2

You want to duplicate a Boolean column and change its title:

```
C_POINTER($ptr)
LISTBOX DUPLICATE COLUMN(*;"boolCol";3;"duplBoolCol";$ptr;"duplBoolHeader";$ptr;"duplBoolFooter";$ptr)
colprt:=OBJECT Get pointer(Object named;"duplBoolCol")
ARRAY BOOLEAN(colprt->;10)
headprt:=OBJECT Get pointer(Object named;"duplBoolHeader")
OBJECT SET TITLE(headprt->;"New duplicated column")
```

# LISTBOX EXPAND

LISTBOX EXPAND ( {* ;} object {; recursive {; selector {; line {; column}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or Variable (if * is omitted) |
| recursive | Boolean | ➡ | True = expand sublevels False = do not expand sublevels |
| selector | Longint | ➡ | Part of list box to expand |
| line | Longint | ➡ | Number of break row to expand or<br>Number of list box level to expand |
| column | Longint | ➡ | Number of break column to expand |

## Description

The **LISTBOX EXPAND** command is used to expand the break rows of the list box object designated by the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string.

If the list box is not configured in hierarchical mode, the command does nothing. For more information about hierarchical list boxes, please refer to the **Managing Hierarchical List Boxes** section.

The optional *recursive* parameter is used to configure the expanding of the hierarchical sublevels of the list box. Pass True or omit this parameter for the command to expand all the levels and all the sublevels. If you pass False, only the first level specified will be expanded.

The optional *selector* parameter is used to specify the scope of the command. You can pass one of the following constants, found in the **List Box** theme, in this parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk all | Longint | 0 | The command affects all sub-levels (default value, used when parameter is omitted). |
| lk selection | Longint | 1 | The command affects selected sub-levels. |
| lk break row | Longint | 2 | The command affects the sub-level to which the "cell" designated by the *row* and *column* parameters belongs. Note that these parameters represent the row and column numbers in the list box in standard mode and not in its hierarchical representation. If the *row* and *column* parameters are omitted, the command does nothing. |
| lk level | Longint | 3 | The command affects all the break rows corresponding to the *level* column. This parameter designates a column number in the list box in standard mode and not in its hierarchical representation. If the *level* parameter is omitted, the command does nothing. |

The command does not select break rows.

If the selection or list box does not contain a break row or if all the break rows are already expanded, the command does nothing.

## Example

This example illustrates different ways of using the command. Given the following arrays shown in a list box:

| | | | |
|---|---|---|---|
| France | Brittany | Brest | 120000 |
| France | Brittany | Quimper | 80000 |
| France | Brittany | Rennes | 200000 |
| France | Normandy | Caen | 220000 |
| France | Normandy | Deauville | 4000 |
| France | Normandy | Cherbourg | 41000 |
| Belgium | Wallonia | Namur | 111000 |
| Belgium | Wallonia | Liege | 200000 |
| Belgium | Flanders | Antwerp | 472000 |
| Belgium | Flanders | Louvain | 95000 |

```
//Expand all the break rows and subrows of the list box
LISTBOX EXPAND(*;"MyListbox")
```

| | |
|---|---|
| V France | |
| V Brittany | |
| Brest | 120000 |
| Quimper | 80000 |
| Rennes | 200000 |
| V Normandy | |
| Caen | 220000 |
| Deauville | 4000 |
| Cherbourg | 41000 |
| V Belgium | |
| V Wallonia | |
| Namur | 111000 |
| Liege | 200000 |
| V Flanders | |
| Antwerp | 472000 |
| Louvain | 95000 |

```
//Expand the first level of break rows of the selection
LISTBOX EXPAND(*;"MyListbox";False;lk selection)
//If the "Belgium" row was selected
```

| |
|---|
| > France |
| V Belgium |
| > Wallonia |
| > Flanders |

```
//Expand the Brittany break row with recursion
LISTBOX EXPAND(*;"MyListbox";False;lk break row;1;2)
```

| | |
|---|---|
| V France | |
| V Brittany | |
| Brest | 120000 |
| Quimper | 80000 |
| Rennes | 200000 |
| > Normandy | |
| > Belgium | |

```
//Expand all the first columns (countries) without recursion
LISTBOX EXPAND(*;"MyListbox";False;lk level;1)
```

| |
|---|
| V France |
| > Brittany |
| > Normandy |
| V Belgium |
| > Wallonia |
| > Flanders |

# LISTBOX Get array

LISTBOX Get array ( {* ;} object ; arrType ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Variable (if * is omitted) |
| arrType | Longint | ⇒ | Type of array |
| Function result | Pointer | ⇐ | Pointer to array associated with property |

## Description

**Note:** This command only works with array type list boxes.

The **LISTBOX Get array** command returns a pointer to the *arrType* array of the list box or list box column designated by the *object* and * parameters.

Style, color, background color or row control arrays can be associated with array type list boxes or (except for row control arrays) with the columns of array type list boxes, using the Property list in Design mode or using the **LISTBOX SET ARRAY** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. You can designate a list box or a list box column in the *object* parameter.

In *arrType*, pass the type of array for the property you want to get. You can use one of the following constants, available in the "**List Box**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk background color array | Longint | 1 | |
| lk control array | Longint | 3 | |
| lk font color array | Longint | 0 | |
| lk row height array | Longint | 4 | (4D View Pro license required) |
| lk style array | Longint | 2 | |

The command returns one of the following values:

- **Nil** if no array for the requested property is associated with the column or the list box.
- a pointer to the array of the requested property, defined by the user.
- a pointer to the array of the requested property, defined dynamically when calling the **LISTBOX SET ROW COLOR** or **LISTBOX SET ROW FONT STYLE** command.

## Example

Typical examples of use:

```
vPtr:=LISTBOX Get array(*;"MyLB";lk font color array)
 // returns a pointer to the font color array
 // associated with the "MyLB" list box

vPtr:=LISTBOX Get array(*;"Col4";lk style array)
 // returns a pointer to the font style array
 // associated with the columns of the "Col4" list box
```

# LISTBOX GET ARRAYS

LISTBOX GET ARRAYS ( {* ;} object ; arrColNames ; arrHeaderNames ; arrColVars ; arrHeaderVars ; arrColsVisible ; arrStyles {; arrFooterNames ; arrFooterVars} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| arrColNames | String array | ⇐ | Column object names |
| arrHeaderNames | String array | ⇐ | Header object names |
| arrColVars | Pointer array | ⇐ | Pointers to column variables or Pointers to column fields or Nil |
| arrHeaderVars | Pointer array | ⇐ | Pointers to header variables |
| arrColsVisible | Boolean array | ⇐ | Visibility of each column |
| arrStyles | Pointer array | ⇐ | Pointers to arrays, or style, color and row control variables, or Nil |
| arrFooterNames | String array | ⇐ | Column footer object names |
| arrFooterVars | Pointer array | ⇐ | Pointers to column footer variables |

## Description

The **LISTBOX GET ARRAYS** command returns a set of synchronized arrays providing information on each column (visible or invisible) in the list box set in the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

Once the command is executed:

- The *arrColNames* array contains the list of object names for each column in the list box.
- The *arrHeaderNames* array contains the list of object names for each column header in the list box.
- The *arrColVars* array contains, for an array type list box, pointers toward variables (arrays) associated with each column of the list box. For a selection type list box, *arrColVars* contains:
  - For a column associated with a field, a pointer to the associated field,
  - For a column associated with a variable, a pointer to the variable,
  - For a column associated with an expression, a Nil pointer.
- The *arrHeaderVars* array contains pointers toward variables associated with each column header of the list box.
- The *arrColsVisible* array contains a Boolean value for each column, indicating whether the column is visible (**True**) or hidden (**False**) in the list box.
- The *arrStyles* array contains, for an array type list box, four pointers to four arrays that allow the applying of a specific style, font color, background color and custom display control to each row of the list box. These arrays are associated with the list box in the Property List of the Design environment or using the **LISTBOX SET ARRAY** command. If an array is not specified for the list box, the corresponding item in *arrStyles* will contain a Nil pointer.
  The fourth pointer corresponds either to a Boolean array (Hidden Rows Array), or to a longint array (array used to set hidden, disabled and non-selectable rows) based on the implementation used for the row control array (see **List box specific properties**).
  For a selection type list box, *arrStyles* contains:
  - For each configuration set via a variable, a pointer to the variable,
  - For each configuration set via an expression, a Nil pointer.

# LISTBOX GET CELL COORDINATES

LISTBOX GET CELL COORDINATES ( {* ;} object ; column ; row ; left ; top ; right ; bottom )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified = object is the name of the object (string)<br>If omitted = object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or variable (if * is omitted) |
| column | Longint | ⇒ | Column number |
| row | Longint | ⇒ | Row number |
| left | Longint | ⇐ | Left coordinate of the object |
| top | Longint | ⇐ | Top coordinate of the object |
| right | Longint | ⇐ | Right coordinate of the object |
| bottom | Longint | ⇐ | Bottom coordinate of the object |

## Description

The **LISTBOX GET CELL COORDINATES** command returns in variables or fields the *left*, *top*, *right* and *bottom* coordinates (in points) of the cell designated by the *column* and *row* parameters, in the list box defined by * and *object*.

If you pass the optional * parameter, it indicates that the *object* parameter is an object name (a string). If you don't pass the optional * parameter, this indicates that the object is a field or a variable. In this case, you pass a field or variable reference (only a field or variable of the object type) instead of a string

For consistency with the **OBJECT GET COORDINATES** command, the origin is the upper-left corner of the form which contains the cell. Also, the coordinates returned are theoretical; they take into account the scrolling state of the list box before any clipping occurs. As a result, the cell may be not visible (or only partially so) at its coordinates, and these coordinates may be outside the form limits (or even negative). To find out if the cell is displayed (and which part of it is visible) you need to compare the coordinates returned with the list box coordinates, while considering the following rules:

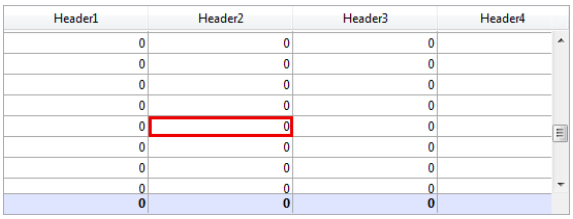
- All cells are clipped to the coordinates of their parent list box (as returned by the **OBJECT GET COORDINATES** command for the list box).
- Header and footer sub-objects are displayed on top column content: when the coordinates of a cell intersect the coordinates of the header or footer rows, then the cell is not displayed at this intersection.
- Elements of locked columns are displayed on top of elements of scrollable columns: when the coordinates of an element in a scrollable column intersect the coordinates of an element in a locked column, then it is not displayed at this intersection.

For more information, please refer to the **OBJECT GET COORDINATES** command description.

## Example

You want to draw a red rectangle around the selected cell of a list box:

```
OBJECT SET VISIBLE(*;"RedRect";False) //initialize a red rectangle
 //the rectangle is already defined somewhere in the form
LISTBOX GET CELL POSITION(*;"LB1";$col;$raw)
LISTBOX GET CELL COORDINATES(*;"LB1";$col;$raw;$x1;$y1;$x2;$y2)
OBJECT SET VISIBLE(*;"RedRect";True)
OBJECT SET COORDINATES(*;"RedRect";$x1;$y1;$x2;$y2)
```



# LISTBOX GET CELL POSITION

LISTBOX GET CELL POSITION ( {* ;} object ; column ; row {; colVar} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| column | Longint | ⇐ | Column number |
| row | Longint | ⇐ | Row number |
| colVar | Pointer | ⇐ | Pointer to column variable |

## Description

The **LISTBOX GET CELL POSITION** command returns the numbers of the *column* and the *row* that correspond to the location of the last mouse click or the last selection made via the keyboard in the listbox designated by * and *object*.

This command returns the coordinates of a click or a selection action even when data entry is not allowed in the list box.

**Notes:**

- The number returned in the *row* parameter does not take into account any hidden/displayed states of list box rows.
- If a cell in a fake column is clicked, the *row* parameter contains "X+1", where X is the number of existing columns. (A fake column can be added automatically when the "Column Auto-Resizing" option is selected; for more information refer to the **Resizing Options theme** paragraph).

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (a string). If you omit this parameter, you indicate that the *object* parameter is a variable.

The optional *colVar* parameter returns a pointer to the variable (i.e. array) associated with the column.

This command can only be called in the framework of a list box that generates one of the following form events:

- On Clicked and On Double Clicked
- On Before Keystroke and On After Keystroke
- On After Edit
- On Getting Focus and On Losing Focus
- On Data Change
- On Selection Change
- On Before Data Entry

When the command is called outside of this context, **LISTBOX GET CELL POSITION** returns 0 in both *column* and *row*.

This command takes into account any selection or deselection actions whether by mouse click, via keyboard keys, or using the **EDIT ITEM** command (which can generate the On Getting Focus event). If the selection is modified using the arrow keys of the keyboard, *column* returns 0. In this case, if it is passed, the *colVar* parameter returns **Nil**.

# LISTBOX Get column formula

LISTBOX Get column formula ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| Function result | String | ⇐ | Formula associated with column |

## Description

The **LISTBOX Get column formula** command returns the formula associated with the list box column designated by the *object* and * parameters. Formulas can only be used when the “Data Source” property of the list box is either **Current Selection** or **Named Selection**. If no formula is associated with the column, the command returns an empty string.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string. This parameter must designate a column of the list box.

# LISTBOX Get column width

LISTBOX Get column width ( {* ;} object {; minWidth {; maxWidth}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| minWidth | Longint | ⇐ | Minimum column width (in pixels) |
| maxWidth | Longint | ⇐ | Maximum column width (in pixels) |
| Function result | Longint | ⇒ | Column width (in pixels) |

## Description

The **LISTBOX Get column width** command returns the width (in pixels) of the column set in the *object* and * parameters. You can pass either a list box column or a column header in the *object* parameter.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

**LISTBOX Get column width** can return the resizing limits of the column in the *minWidth* and *maxWidth* parameters. These limits can be specified via the **LISTBOX SET COLUMN WIDTH** command.
If no minimum and/or maximum value has been set for the column, the corresponding parameter returns 0.

# LISTBOX Get footer calculation

LISTBOX Get footer calculation ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| Function result | Longint | ⇐ | Type of calculation |

## Description

The **LISTBOX Get footer calculation** command returns the type of calculation associated with the footer area of the list box designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

The *object* parameter can designate:

- the variable or name of a footer area. In this case, the command returns the calculation associated with this area.
- the variable or name of a list box column. In this case, the command returns the calculation associated with footer area of this column.

You can compare the value returned with the constants of the **Listbox Footer Calculation** theme (see the **LISTBOX SET FOOTER CALCULATION** command).

# LISTBOX Get footers height

LISTBOX Get footers height ( {* ;} object {; unit} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| unit | Longint | ⇒ | Unit of height value:<br>0 or omitted = pixels, 1 = lines |
| Function result | Longint | ⇐ | Row height |

## Description

The **LISTBOX Get footers height** command returns the height of the footer row in the list box designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string. You can designate either the list box or any footer of the list box.

By default, if you omit the *unit* parameter, the height of the row returned is expressed in pixels. To set a different unit, you can pass one of the following constants (found in the **List Box** theme), in the *unit* parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk lines | Longint | 1 | Height designates a number of lines. 4D calculates the height of a line according to the font |
| lk pixels | Longint | 0 | Height is a number of pixels (default). |

**Note:** For more information about calculating the height of rows, refer to the *Design Reference* manual.

## LISTBOX GET GRID

LISTBOX GET GRID ( {* ;} object ; horizontal ; vertical )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| horizontal | Boolean | ⇐ | True = shown, False = hidden |
| vertical | Boolean | ⇐ | True = shown, False = hidden |

### Description

The **LISTBOX GET GRID** command returns the shown/hidden status of the horizontal and/or vertical lines making up the grid of the list box object designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

In *horizontal* and *vertical*, the command returns the value **True** or **False** depending on whether the corresponding lines are shown (True) or hidden (False). By default, the grid is shown.

# LISTBOX GET GRID COLORS

LISTBOX GET GRID COLORS ( {* ;} object ; hColor ; vColor )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| hColor | Longint | ⇐ | Value of RGB color for horizontal lines |
| vColor | Longint | ⇐ | Value of RGB color for vertical lines |

## Description

The **LISTBOX GET GRID COLORS** command returns the color of the horizontal and vertical lines making up the grid of the list box object designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

In *hColor* and *vColor*, the command returns the values of the RGB colors. For more information about RGB colors, refer to the description of the **OBJECT SET RGB COLORS** command.

# LISTBOX Get headers height

LISTBOX Get headers height ( {* ;} object {; unit} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| unit | Longint | ⇒ | Unit of height value:<br>0 or omitted = pixels, 1 = lines |
| Function result | Longint | ⇐ | Row height |

## Description

The **LISTBOX Get headers height** command returns the height of the header row in the list box designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string. You can designate either the list box or any header of the list box.

By default, if you omit the *unit* parameter, the height of the row returned is expressed in pixels. To set a different unit, you can pass one of the following constants (found in the **List Box** theme), in the *unit* parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk lines | Longint | 1 | Height designates a number of lines. 4D calculates the height of a line according to the font |
| lk pixels | Longint | 0 | Height is a number of pixels (default). |

**Note:** For more information about calculating the height of rows, refer to the *Design Reference* manual.

# LISTBOX GET HIERARCHY

LISTBOX GET HIERARCHY ( {* ;} object ; hierarchical {; hierarchy} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| hierarchical | Boolean | ⇐ | True = hierarchical list box False = non-hierarchical list box |
| hierarchy | Pointer array | ⇐ | Array of pointers |

## Description

The **LISTBOX GET HIERARCHY** command lets you find out the hierarchical properties of the list box object designated by the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string.

The Boolean *hierarchical* parameter indicates whether or not the list box is in hierarchical mode:

- if the parameter returns True, the list box is in hierarchical mode,
- if the parameter returns False, the list box is displayed in non-hierarchical mode (standard array mode).

If the list box is in hierarchical mode, the command fills the *hierarchy* array with pointers to the arrays of the list box used to set the hierarchy.

**Note:** If the list box is in non-hierarchical mode, the command returns, in the first element of the *hierarchy* array, a pointer to the array of the first column of the list box.

# LISTBOX Get information

LISTBOX Get information ( {* ;} object ; info ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string). If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| info | Longint | ⇒ | Information to get |
| Function result | Longint | ⇐ | Current value |

## Description

The **LISTBOX Get information** command returns various information regarding the current visibility and size of headers, footers and scrollbars in the list box object set using the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

In the *info* parameter, pass a value indicating the type of information that you want to get. You can use a value or one of the following constants from the "**List Box**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk display footer | Longint | 8 | **Display Footers** property<br>Applies to: List box<br>Possible values:<br>• lk no (0): hidden<br>• lk yes (1): shown |
| lk display header | Longint | 0 | **Display Headers** property<br>Applies to: List box<br>Possible values:<br>• lk no (0): hidden<br>• lk yes (1): shown |
| lk display hor scrollbar | Longint | 2 | 0=hidden, 1=shown |
| lk display ver scrollbar | Longint | 4 | 0=hidden, 1=shown |
| lk footer height | Longint | 9 | Height in pixels |
| lk header height | Longint | 1 | Height in pixels |
| lk hor scrollbar height | Longint | 3 | Height in pixels |
| lk hor scrollbar position | Longint | 6 | Position of the cursor in pixels |
| lk ver scrollbar position | Longint | 7 | Position of the cursor in pixels |
| lk ver scrollbar width | Longint | 5 | Width in pixels |

- The first constants are useful for calculating the actual size of a list box area in a form.
- When you use the constants lk hor scrollbar position or lk ver scrollbar position, the **LISTBOX Get information** command returns the position of the scrolling cursor in relation to its original position, i.e. the size of the hidden part of the window, expressed in pixels. By default, this position corresponds to 0. Combined, for example, with information concerning the row height, this value lets you find out the contents displayed in the listbox.
- The statement **LISTBOX Get information**(vLB;lk footer height) returns the same value as the **LISTBOX Get footers height** command when footers are displayed. However, if footers are not displayed, **LISTBOX Get information** returns 0 while **LISTBOX Get footers height** still returns the height, in this case theoretical, of the footers.

## Example

Given a list box containing rows with a height of 20 pixels each. You execute the following statement:

```
$scroll:=LISTBOX Get information(*;"Listbox";lk ver scrollbar position)
```

If, for instance, $scroll returns 200, you can conclude that the 11th row is currently the first one displayed in the list box (200/20=10, thus 10 rows are hidden).

# LISTBOX Get locked columns

LISTBOX Get locked columns ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| Function result | Longint | ⇐ | Number of locked columns |

## Description

The **LISTBOX Get locked columns** command returns the number of locked columns in the list box designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

Columns can be locked through the Property List or using the **LISTBOX SET LOCKED COLUMNS** command. For more information, refer to the *Design Reference* manual.

If a column is inserted or deleted by programming within the locked area, the number of columns returned by this command takes this change into account. For example, if you delete a locked column, the number of locked columns is decreased by 1. Similarly, if you insert a column by programming into a locked area, this column is locked automatically and the number of locked columns is increased by 1.

However, the command does not take into account the visible/invisible status of columns.

# LISTBOX Get number of columns

LISTBOX Get number of columns ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | Longint | ⇐ | Number of columns |

## Description

The **LISTBOX Get number of columns** command returns the total number of columns (visible or invisible) present in the list box set in the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information on object names, refer to the **Object Properties** section.

# LISTBOX Get number of rows

LISTBOX Get number of rows ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | Longint | ⇐ | Number of rows |

## Description

The **LISTBOX Get number of rows** command returns the number of rows in the list box set in the *object* and * parameters.

**Note:** This command does not take the hidden/displayed state of the rows into account. For example, in a list box with 10 rows where the first 9 rows are hidden, **LISTBOX Get number of rows** will return 10.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

**Note:** If the arrays associated with the columns of a List box do not all have the same size, only the number of items corresponding to the smallest array will appear in the list box and thus be returned by this command.

# LISTBOX GET OBJECTS

LISTBOX GET OBJECTS ( {* ;} object ; arrObjectNames )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| arrObjectNames | Text array | ⇐ | Names of sub-objects comprising list box (headers, columns, footers) |

## Description

The **LISTBOX GET OBJECTS** command returns an array containing the names of each object making up the list box designated by the *object* and * parameters.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the the *object* parameter is a variable. In this case, you pass a variable reference instead of a string.

In *arrObjectNames*, you pass a text array that is automatically filled in by the command. Object names are returned in their display order, with the following sequence:

```
nameCol1
headerNameCol1
footerNameCol1
nameCol2
headerNameCol2
footerNameCol2
...
```

The array returns the object names for all the columns (including column footers), regardless of whether or not they are visible.

This command is useful in the context of the parsing of a form using the **FORM LOAD**, **FORM GET OBJECTS** and **OBJECT Get type** commands. You can use it, when needed, to obtain the names of list box sub-objects.

## Example

You want to load a form and get a list of all the objects of list boxes that it contains.

```
FORM LOAD("MyForm")
ARRAY TEXT(arrObjects;0)
FORM GET OBJECTS(arrObjects)
ARRAY LONGINT(ar_type;Size of array(arrObjects))
For($i;1;Size of array(arrObjects))
   ar_type{$i}:=OBJECT Get type(*;arrObjects{$i})
   If(ar_type{$i}=Object type listbox)
      ARRAY TEXT(arrLBObjects;0)
      LISTBOX GET OBJECTS(*;arrObjects{$i};arrLBObjects)
   End if
End for
FORM UNLOAD
```

# LISTBOX GET PRINT INFORMATION

LISTBOX GET PRINT INFORMATION ( {* ;} object ; selector ; info )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| selector | Longint | ⇒ | Information to get |
| info | Longint | ⇐ | Current value |

## Description

The **LISTBOX GET PRINT INFORMATION** command returns the current information relative to the printing of the list box object designated by the *object* and * parameters. This command can be used to control the printing of the list box contents.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string.

This command must be called in the context of the printing of a list box via the **Print object** command. Outside of this context, it will not return significant values.

Pass a value indicating the information you want to find out in *selector* and a variable of the number or BLOB type in *info*. In *selector*, you can pass one of the following constants, found in the "**List Box**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk last printed row number | Longint | 0 | Returns in *info* the number of the last row printed. Lets you find out the number of the next row to be printed.<br>The number returned may be greater than the number of rows actually printed if the list box contains invisible rows or if the **OBJECT SET SCROLL POSITION** command has been called. For example, if rows 1, 18 and 20 have been printed, *info* is 20. |
| lk printed height | Longint | 3 | Returns in *info* the height in pixels of the object actually printed (including headers, lines, etc.). Remember that if the number of rows to print is less than the "capacity" of the list box, its height is automatically reduced. |
| lk printed rows | Longint | 1 | Returns in *info* the number of rows actually printed during the last call to the **Print object** command. This number includes any break rows added in the case of a hierarchical list box. For example, *info* is 10 if the list box contains 20 rows and the odd-numbered rows were hidden. |
| lk printing is over | Longint | 2 | Returns in *info* a Boolean indicating whether the last (visible) row of the list box has actually been printed. True = row has been printed; Otherwise, False. |

For more information about the principles of printing list boxes, please refer to **Printing list boxes**.

## Example 1

Printing until all the rows have been printed:

```
OPEN PRINTING JOB
FORM LOAD("SalesForm")

$Over:=False
Repeat
   $Total:=Print object(*;"mylistbox")
   LISTBOX GET PRINT INFORMATION(*;"mylistbox";lk printing is over;$Over)
   PAGE BREAK
Until($Over)

CLOSE PRINTING JOB
```

## Example 2

Printing at least 500 rows of the list box, knowing that certain rows are hidden:

```
$GlobalPrinted:=0
Repeat
   $Total:=Print object(*;"mylistbox")
   LISTBOX GET PRINT INFORMATION(*;"mylistbox";lk printed rows;$Printed)
   $GlobalPrinted:=$GlobalPrinted+$Printed
   PAGE BREAK
Until($GlobalPrinted>=500)
```

# LISTBOX Get row color

LISTBOX Get row color ( {* ;} object ; row {; colorType} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Variable (if * is omitted) |
| row | Longint | ⇒ | Row number |
| colorType | Longint | ⇒ | Listbox font color (default) or Listbox background color |
| Function result | Longint | ⇐ | Color value |

## Description

**Note:** This command only works with array type list boxes.

The **LISTBOX Get row color** command returns the color of a row or a cell in the list box designated by the *object* and * parameters.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. You can designate a list box or a list box column in the *object* parameter:

- When *object* designates a list box, the command returns the color of the row.
- When *object* designates a list box column, the command returns the color of the cell.

In *row*, pass the number of the row whose color you want to get.

**Note:** The command does not take any hidden/shown states of the list box rows into account.

In the *colorType* parameter, you can pass either the lk background color or lk font color constant ("**List Box**" theme) in order to find out the background or font color for the row. If you omit this parameter, the font color is returned.

**Warning:** a color assigned to a row is not necessarily displayed in every cell of the row (see example). If conflicting color values are set using properties for list boxes or list box columns, an order of priority is applied. For more information, refer to the *Design Reference* manual.

## Example

Given the following list box:

| | | | | | |
|---|---|---|---|---|---|
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |

```
vColor:=LISTBOX Get row color(*;"Col5";3)
vColor2:=LISTBOX Get row color(*;"List Box";3)
vColor3:=LISTBOX Get row color(*;"List Box";lk background color)
 // vColor contains 0xFFFF00 (yellow)
 // vColor2 contains 0x00FF (blue)
 // vColor3 contains 0x00FF0000 (red)
```

# LISTBOX Get row font style

LISTBOX Get row font style ( {* ;} object ; row ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Variable (if * is omitted) |
| row | Longint | ⇒ | Row number |
| Function result | Longint | ⇐ | Style value |

## Description

**Note:** This command only works with array type list boxes.

The **LISTBOX Get row font style** command returns the font style of a row or a cell in the list box designated by the *object* and * parameters.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. You can designate a list box or a list box column in the *object* parameter:

- When *object* designates a list box, the command returns the style of the row.
- When *object* designates a list box column, the command returns the style of the cell.

In *row*, pass the number of the row whose style you want to get.

**Note:** The command does not take any hidden/shown states of the list box rows into account.

**Warning:** a style assigned to a row is not necessarily displayed in every cell of the row (see example). If conflicting color values are set using properties for list boxes or list box columns, an order of priority is applied. For more information, refer to the *Design Reference* manual.

## Example

Given the following list box:

| | | | | | |
|---|---|---|---|---|---|
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| *text* | *text* | *text* | *text* | **text** | *text* |
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |

```
vStyle:=LISTBOX Get row font style(*;"Col5";3)
vStyle2:=LISTBOX Get row font style(*;"List Box";3)
 // vStyle contains 1 (Bold)
 // vStyle2 contains 6 (Italic + Underline)
```

# LISTBOX Get row height

LISTBOX Get row height ( {* ;} object ; row ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| row | Longint | ⇒ | List box row whose height you want to get |
| Function result | Integer | ⇐ | Row height |

## 4D View Pro

This command requires a 4D View Pro license. If this license is not available, an error is displayed in the list box when the form is executed. For more information, please refer to the **4D View Pro** section.

## Description

The **LISTBOX Get row height** command returns the current height of the specified *row* in the list box object designated using the *object* and * parameters. Row height can be set globally using the Property List or the **LISTBOX SET ROWS HEIGHT** command, or individually by means of the **LISTBOX SET ROW HEIGHT** command.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

If the specified *row* does not exist in the list box, the command returns 0 (zero).

The row height is returned using the current unit defined for the list box rows globally, either in the Property list or by a prior call to the **LISTBOX SET ROWS HEIGHT** command.

# LISTBOX Get rows height

LISTBOX Get rows height ( {* ;} object {; unit} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| unit | Longint | ⇒ | Unit of height value: 0 or omitted = pixels, 1 = lines |
| Function result | Integer | ⇐ | Row height |

## Description

The **LISTBOX Get rows height** command returns the current row height for the list box object set using the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

By default, if you omit the *unit* parameter, the row height returned is expressed in pixels. To set another unit, in the *unit* parameter you can pass one of the following constants, found in the **List Box** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk lines | Longint | 1 | Height designates a number of lines. 4D calculates the height of a line according to the font |
| lk pixels | Longint | 0 | Height is a number of pixels (default). |

**Note:** For more information about calculating the height of rows, refer to the *Design Reference* manual.

# LISTBOX Get static columns

LISTBOX Get static columns ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| Function result | Longint | ⇐ | Number of static columns |

## Description

The **LISTBOX Get static columns** command returns the number of static columns in the list box designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

Static columns can be set through the Property List or using the **LISTBOX SET STATIC COLUMNS** command.

If a column is inserted or deleted by programming within a set of static columns, the number of columns returned by this command takes this change into account.
However, the command does not take into account the visible/invisible status of columns.

**Note:** Static columns and locked columns are two independent functions. For more information, refer to the *Design Reference* manual.

# LISTBOX GET TABLE SOURCE

LISTBOX GET TABLE SOURCE ( {* ;} object ; tableNum {; name {; highlightName}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| tableNum | Longint | ⇐ | Table number of selection |
| name | String | ⇐ | Name of named selection or "" for the current selection |
| highlightName | String | ⇐ | Name of highlight set |

## Description

The **LISTBOX GET TABLE SOURCE** command can be used to find out the current source of the data displayed in the list box that is designated by the * and *object* parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, please refer to the **Object Properties** section.

The command returns the number of the main table associated with the list box in the *tableNum* parameter and the name of any named selection used in the optional *name* parameter.

If the rows of the list box are linked with the current selection of the table, the *name* parameter, if passed, returns an empty string. If the rows of the list box are linked with a named selection, the *name* parameter returns the name of this named selection.

If the list box is associated with arrays, *tableNum* returns -1 and *name*, if passed, returns an empty string.

# LISTBOX INSERT COLUMN

LISTBOX INSERT COLUMN ( {* ;} object ; colPosition ; colName ; colVariable ; headerName ; headerVar {; footerName ; footerVar} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is set) or Variable (if * is omitted) |
| colPosition | Longint | ⇒ | Location of column to insert |
| colName | String | ⇒ | Name of the column object |
| colVariable | Array, Field, Variable, Nil pointer | ⇒ | Column array name or field or variable |
| headerName | String | ⇒ | Name of the column header object |
| headerVar | Integer variable, Nil pointer | ⇒ | Column header variable |
| footerName | String | ⇒ | Column footer object name |
| footerVar | Variable, Nil pointer | ⇒ | Column footer variable |

## Description

The **LISTBOX INSERT COLUMN** command inserts a column in the list box set by the *object* and * parameters.

**Note:** This command does nothing if it is applied to the first column of a list box displayed in hierarchical mode.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

The new column is inserted just in front of the column set using the *colPosition* parameter. If the *colPosition* parameter is greater than the total number of columns, the column is added after the last column.

Pass the name of the object and the variable of the inserted column in the *colName* and *colVariable* parameters.

- With an array type list box, the name of the variable must match the name of the array whose contents will be displayed in the column. You can pass a Nil (->[]) pointer if you use the command in a dynamic context when the form is executed (see below).
- With a selection type list box, you must pass a field or variable in the *colVariable* parameter. The contents of the column will thus be the value of the field or variable, evaluated for each record of the selection associated with the list box. This type of contents can only be used when the “Data Source” property of the list box is Current Selection or Named Selection (see the **Managing List Box Objects** section). You can use fields or variables of the string, number, Date, Time, Picture and Boolean types.

In the context of list boxes based on selections, **LISTBOX INSERT COLUMN** can be used to insert simple elements (fields or variables). If you want to handle more complex expressions (such as formulas or methods), you must use the **LISTBOX INSERT COLUMN FORMULA** command.

**Note:** It is not possible to combine columns of the array type (array data source) and those of the field or variable type (selection data source) in the same list box.

Pass the object name and the variable of the inserted column header in the *headerName* and *headerVar* parameters.

In the *footerName* and *footerVar* parameters, you can also pass the object name and variable of the footer of the inserted column.

**Note:** Object names must be unique in a form. You must be sure that the names passed in the *colName*, *headerName* and *footerName* parameters are not already used. Otherwise, the column is not created and an error is generated.

### Dynamic insertion

Starting with 4D v14 R3, you can use this command to insert columns into list boxes dynamically when the form is executed. 4D will automatically handle the definition of the necessary variables (column, footer and header).

To do this, **LISTBOX INSERT COLUMN** accepts a **Nil** (**->[]**) pointer as a value for the *colVariable* (array type list box only), *headerVar* and *footerVar* parameters. In this case, when the command is executed, 4D creates the required variables dynamically (for more information, refer to the **Dynamic variables** section).

Note that header and footer variables are always created with a specific type (longint and text, respectively). Conversely, column variables cannot be typed when created because list boxes accept different types of arrays for these variables (text array, integer array, and so on). This means you have to set the array type manually (see example 3). It is important to

perform this typing before calling commands such as **LISTBOX INSERT ROWS** to insert new elements in the array. Alternatively, you can use **APPEND TO ARRAY** both for setting the type of the array and inserting elements.

## Example 1

We would like to add a column at the end of the list box:

```
C_LONGINT(HeaderVarName;$Last;RecNum)
ALL RECORDS([Table 1])
$RecNum:=Records in table([Table 1])
ARRAY PICTURE(Picture;$RecNum)

$Last:=LISTBOX Get number of columns(*;"ListBox1")+1
LISTBOX INSERT COLUMN(*;"ListBox1";$Last;"ColumnPicture";Picture;"HeaderPicture";HeaderVarName)
```

## Example 2

We would like to add a column to the right of the list box and associate the values of the [Transport]Fees field with it:

```
$last:=LISTBOX Get number of columns(*;"ListBox1")+1
LISTBOX INSERT COLUMN(*;"ListBox1";$last;"FieldCol";[Transport]Fees;"HeaderName";HeaderVar)
```

## Example 3

You want to insert a column dynamically into an array type list box and define its header:

```
C_POINTER($NilPtr)
LISTBOX INSERT COLUMN(*;"MyListBox";1;"MyNewColumn";$NilPtr;"MyNewHeader";$NilPtr)
ColPtr:=OBJECT Get pointer(Object named;"MyNewColumn")
ARRAY TEXT(ColPtr->;10)
 //Definition of header
headprt:=OBJECT Get pointer(Object named;"MyNewHeader")
OBJECT SET TITLE(headprt->;"Inserted header")
```

# LISTBOX INSERT COLUMN FORMULA

LISTBOX INSERT COLUMN FORMULA ( {* ;} object ; colPosition ; colName ; formula ; dataType ; headerName ; headerVar {; footerName ; footerVar} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| colPosition | Longint | ⇒ | Location of column to insert |
| colName | String | ⇒ | Name of the column object |
| formula | String | ⇒ | 4D formula associated with column |
| dataType | Longint | ⇒ | Type of formula result |
| headerName | String | ⇒ | Name of the column header object |
| headerVar | Integer variable, Nil pointer | ⇒ | Column header variable |
| footerName | String | ⇒ | Column footer object name |
| footerVar | Variable, Nil pointer | ⇒ | Column footer variable |

## Description

The **LISTBOX INSERT COLUMN FORMULA** command inserts a column into the listbox designated by the *object* and * parameters.

The **LISTBOX INSERT COLUMN FORMULA** command is similar to the **LISTBOX INSERT COLUMN** command except that it can be used to enter a formula as the contents of a column.

This type of contents can only be used when the "Data Source" property of the list box is set to **Current Selection** or **Named Selection** (for more information about this, please refer to the **Managing List Box Objects** section).

**Note:** This command does nothing if it is applied to the first column of a list box displayed in hierarchical mode.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, please refer to the **Object Properties** section.

The new column is inserted just before the column designated by the *colPosition* parameter. If the *colPosition* parameter is greater than the total number of columns, the column will be added after the last column.

Pass the object name of the inserted column in the *colName* parameter.

The *formula* parameter can contain any valid expression, i.e.:

- An instruction,
- A formula generated using the Formula editor,
- A call to a 4D command,
- A call to a project method.

At the moment the command is called, the *formula* is parsed then executed.

**Note:** Use the **Command name** command in order to define formulas that are independent from the application language (when they call on 4D commands).

The *dataType* parameter can be used to designate the type of data resulting from the execution of the formula. You must pass one of the following constants of the "**Field and Variable Types**" theme in this parameter:

| Constant | Type | Value |
|---|---|---|
| Is Boolean | Longint | 6 |
| Is date | Longint | 4 |
| Is picture | Longint | 3 |
| Is real | Longint | 1 |
| Is text | Longint | 2 |
| Is time | Longint | 11 |

If the result of the *formula* does not correspond to the expected data type, an error is generated.

In the *headerName* and *headerVar* parameters, pass the object name and variable of the column header inserted.

In the *footerName* and *footerVar* parameters, you can also pass the object name and variable of the footer of the inserted column. If you omit the *footerVar* parameter, 4D uses a dynamic variable.

**Note:** Object names must be unique in a form. You need to make sure that the names passed in the *colName*, *headerName* and *footerName* parameters are not already used. Otherwise, the column is not created and an error is generated.

### Dynamic insertion

Starting with 4D v14 R3, you can use this command to insert columns into list boxes dynamically when the form is executed. 4D will automatically handle the definition of the necessary variables (footer and header).

To do this, **LISTBOX INSERT COLUMN FORMULA** accepts a **Nil** (**->[]**) pointer as a value for the *headerVar* and *footerVar* parameters. In this case, when the command is executed, 4D creates the required variables dynamically (for more information, refer to the **Dynamic variables** section).

Note that header and footer variables are always created with a specific type (longint and text, respectively).

## Example

We want to add a new column to the right of the list box that will contain a formula which calculates an empolyee's age:

```
vAge:="Current Date-[Employees]BirthDate)¥365"
$last:=LISTBOX Get number of columns(*;"ListBox1")+1
LISTBOX INSERT COLUMN FORMULA(*;"ListBox1";$last;"ColFormula";vAge;Is real;"Age";HeaderVar)
```

# LISTBOX INSERT ROWS

LISTBOX INSERT ROWS ( {* ;} object ; rowPosition {; numRows} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or Variable (if * is omitted) |
| rowPosition | Longint | ➡ | Position of the row to insert |
| numRows | Longint | ➡ | Number of rows to insert |

## Description

The **LISTBOX INSERT ROWS** command inserts one or several new row(s) in the list box set in the *object* and * parameters.

**Note:** This command only works with list boxes based on arrays. When this command is used with a list box based on a selection, it does nothing and the OK system variable is set to 0.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

By default, if *numRows* is omitted, only one row is inserted. Otherwise, the command inserts the number of rows defined in this parameter.

This command inserts the row(s) at the position set by the *rowPosition* parameter and these row(s) are automatically added at this position in all the arrays used by the list box columns, whatever their type and their visibility.

If the *rowPosition* value is higher than the total number of rows in the list box, the command adds the row(s) at the end of each array. If it is equal to 0, the command adds the row(s) at the beginning of each array. If it contains a negative value, the command does nothing.

# LISTBOX MOVE COLUMN

LISTBOX MOVE COLUMN ( {* ;} object ; colPosition )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or<br>Variable (if * is omitted) of the column to be moved |
| colPosition | Longint | ➡ | New location of column |

## Description

The **LISTBOX MOVE COLUMN** command moves the column designated by the *object* and * parameters by programming in the context of the form being executed (Application mode). The original form, generated in the Design mode, is not modified.

The *object* and * parameters designate the column to move. Passing the optional * parameter indicates that the *object* parameter is a column name (string). If you do not pass this parameter, this indicates that the *object* parameter is a column variable. In this case, you pass a variable reference instead of a string.

The column is moved to just in front of the one designated by the *colPosition* parameter. If the *colPosition* parameter is greater than the total number of columns, then the column is moved to just after the last column.

**Note:** This command does nothing when it is applied to the first column of a list box displayed in hierarchical mode.

The command takes the static and locked column properties into account: for example, if you try to move a static column, the command does nothing.

This functionality was already found in 4D in Application mode: the user can move non-static columns using the mouse. However, unlike columns moved by the user, this command does not generate the On Column Moved event.

## Example

You want to swap the 2nd and 3rd columns of the list box:

```
LISTBOX MOVE COLUMN(*;"column2";3)
```

# LISTBOX MOVED COLUMN NUMBER

LISTBOX MOVED COLUMN NUMBER ( {* ;} object ; oldPosition ; newPosition )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| oldPosition | Longint | ⇐ | Previous position of the moved column |
| newPosition | Longint | ⇐ | New position of the moved column |

## Description

The **LISTBOX MOVED COLUMN NUMBER** command returns two numbers in *oldPosition* and *newPosition* indicating respectively the previous position and the new position of the column moved in the list box, specified by the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

This command must be used with the form event On Column Moved (see the **Form event** command).

**Note:** This command takes invisible columns into account.

# LISTBOX MOVED ROW NUMBER

LISTBOX MOVED ROW NUMBER ( {* ;} object ; oldPosition ; newPosition )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| oldPosition | Longint | ⇐ | Previous position of the moved row |
| newPosition | Longint | ⇐ | New position of the moved row |

## Description

The **LISTBOX MOVED ROW NUMBER** command returns two numbers in *oldPosition* and *newPosition* indicating respectively the previous position and the new position of the row moved in the list box, specified by the *object* and * parameters.

**Note:** You can only move rows in array type list boxes.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

This command must be used with the form event On Row Moved (see the **Form event** command).

**Note:** This command does not take into account any hidden/displayed states of list box rows.

# LISTBOX SELECT BREAK

LISTBOX SELECT BREAK ( {* ;} object ; row ; column {; action} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or Variable (if * is omitted) |
| row | Longint | ➡ | Number of break row |
| column | Longint | ➡ | Number of break column |
| action | Longint | ➡ | Selection action |

## Description

The **LISTBOX SELECT BREAK** command can be used to select break rows in the list box object designated by the *object* and * parameters. The list box must be displayed in hierarchical mode.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string.

Break rows are added to represent the hierarchy but they do not correspond to existing rows in the array. To designate a break row to be selected, in the *row* and *column* parameters, you must pass the row and column number corresponding to the first occurrence in the corresponding array. These values are returned by the **LISTBOX GET CELL POSITION** command when the user has selected a break row. This principle is described in the “Management of break rows” paragraph of the **Managing Hierarchical List Boxes** section.

The *action* parameter, if it is passed, can set the selection action to be carried out when a selection of break rows already exists in the list box. You can pass a value or one of the following constants, found in the "**List Box**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk add to selection | Longint | 1 | The row selected is added to the existing selection. If the row specified already belongs to the existing selection, the command does nothing. |
| lk remove from selection | Longint | 2 | The row selected is removed from the existing selection. If the row specified does not belong to the existing selection, the command does nothing. |
| lk replace selection | Longint | 0 | The row selected becomes the new selection and replaces the existing selection. The command has the same effect as a user click on a row (however, the On Clicked event is not generated). This is the default action (if the *action* parameter is omitted). |

## Example

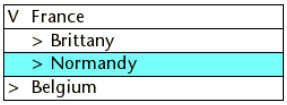
Given the following arrays shown in a list box:

| (T1) | (T2) | (T3) | (T4) |
|---|---|---|---|
| France | Brittany | Brest | 120000 |
| France | Brittany | Quimper | 80000 |
| France | Brittany | Rennes | 200000 |
| France | Normandy | Caen | 220000 |
| France | Normandy | Deauville | 4000 |
| France | Normandy | Cherbourg | 41000 |
| Belgium | Wallonia | Namur | 111000 |
| Belgium | Wallonia | Liege | 200000 |
| Belgium | Flanders | Antwerp | 472000 |
| Belgium | Flanders | Louvain | 95000 |

We want to select the "Normandy" break row:

```
$row:=Find in array(T2;"Normandy")
$column:=2
LISTBOX COLLAPSE(*;"MyListbox") `collapsing of all levels
LISTBOX SELECT BREAK(*;"MyListbox";$row;$column)
```

Here is the result:


V France
> Brittany
> Normandy
> Belgium

# LISTBOX SELECT ROW

LISTBOX SELECT ROW ( {* ;} object ; rowPosition {; action} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| rowPosition | Longint | ⇒ | Number of the row to select |
| action | Longint | ⇒ | Selection action |

## Description

The **LISTBOX SELECT ROW** command selects the row whose number is passed in *position* in the list box set in the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

The optional *action* parameter, if passed, is used to define the selection action to execute when a selection of rows already exists in the list box. You can pass a value or one of the following constants (located in the "**List Box**" theme):

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk add to selection | Longint | 1 | The row selected is added to the existing selection. If the row specified already belongs to the existing selection, the command does nothing. |
| lk remove from selection | Longint | 2 | The row selected is removed from the existing selection. If the row specified does not belong to the existing selection, the command does nothing. |
| lk replace selection | Longint | 0 | The row selected becomes the new selection and replaces the existing selection. The command has the same effect as a user click on a row (however, the On Clicked event is not generated). This is the default action (if the *action* parameter is omitted). |

When the *position* parameter does not correspond exactly to an existing row number, the command works as follows:

- If *position* is <0, the command does nothing, regardless of the *action* parameter value.
- If *position* is 0 and if the *action* parameter contains lk replace selection or is omitted, all the rows of the listbox are selected. If the *action* parameter contains lk remove from selection, all the listbox rows are deselected.
- If the *position* value is greater than the total number of rows contained in the listbox (only in the case of an array type listbox), the Boolean array associated with the listbox is automatically resized and the selection action is carried out. This mechanism means that you can use **LISTBOX SELECT ROW** with "standard" array management commands (such as **APPEND TO ARRAY**) that do not cause immediate synchronization of the listbox.
  After execution of the method, the arrays are synchronized: if the source array of the listbox has indeed been resized, the selection action is carried out. Otherwise, the Boolean array associated with the listbox returns to its initial size and the command does nothing.

**Notes:**

- If you want the list box to scroll automatically in order to display the row selected, use the **OBJECT SET SCROLL POSITION** command.
- To switch a row into editing mode (to allow data entry), use the **EDIT ITEM** command.
- If the number passed in *position* corresponds to a hidden row in the list box, the row is selected but not displayed.

# LISTBOX SET ARRAY

LISTBOX SET ARRAY ( {* ;} object ; arrType ; arrPtr )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or<br>Variable (if * is omitted) |
| arrType | Longint | ➡ | Type of array |
| arrPtr | Pointer | ➡ | Array to associate with property |

## Description

**Note:** This command only works with array type list boxes.

The **LISTBOX SET ARRAY** command associates an *arrType* array with the list box or list box column designated by the *object* and * parameters.

**Note:** Arrays of styles, colors, background colors or row controls can also be associated with array type list boxes using the Property list in the Design mode.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. You can designate a list box or a list box column in the *object* parameter.

In *arrType*, pass the type of array to associate with the list box or column. You can use one of the following constants, available in the "**List Box**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk background color array | Longint | 1 | |
| lk control array | Longint | 3 | |
| lk font color array | Longint | 0 | |
| lk row height array | Longint | 4 | (4D View Pro license required) |
| lk style array | Longint | 2 | |

In the *arrPtr* parameter, you pass a pointer to the array to use to support the property type.

## Example 1

You want to reuse the font color array of the 4th column for the 10th column:

```
 // retrieve a pointer to the array for column 4
$Pointer:=LISTBOX Get array(*;"Col4";lk font color array)
 // check that it exists
If(Not(Nil($Pointer)))
 //transfer to column 10
   LISTBOX SET ARRAY(*;"Col10";lk font color array;$Pointer)
End if
```

## Example 2

You want to set a row height array for a list box:

```
LISTBOX SET ARRAY(*;"LB";lk row height array;->RowHeightArray)
```

**Note**: The **Row Height Array** property for list boxes requires a 4D View Pro license. For more information, refer to **4D View Pro**.

# LISTBOX SET COLUMN FORMULA

LISTBOX SET COLUMN FORMULA ( {* ;} object ; formula ; dataType )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| formula | String | ⇒ | 4D formula associated with column |
| dataType | Longint | ⇒ | Type of formula result |

## Description

The **LISTBOX SET COLUMN FORMULA** command modifies the *formula* associated with the column of the list box designated by the *object* and * parameters. Formulas can only be used when the “Data Source” property of the list box is either **Current Selection** or **Named Selection**.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string. This parameter must designate a list box column.

The *formula* parameter can contain any valid expression, i.e.:

- an instruction,
- a formula generated using the formula editor,
- a call to a 4D command,
- a call to a project method.

When the command is called, the formula is parsed and then executed.

**Note:** Use the **Command name** command to specify formulas independent from the application language (when they call 4D commands).

The *dataType* parameter designates the type of data resulting from the execution of the formula. In this parameter, you pass one of the constants from the **Field and Variable Types** theme. If the formula result does not match the expected data type, an error is generated.

# LISTBOX SET COLUMN WIDTH

LISTBOX SET COLUMN WIDTH ( {* ;} object ; width {; minWidth {; maxWidth}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| width | Longint | ⇒ | Column width (in pixels) |
| minWidth | Longint | ⇒ | Minimum column width (in pixels) |
| maxWidth | Longint | ⇒ | Maximum column width (in pixels) |

## Description

The **LISTBOX SET COLUMN WIDTH** command allows you to modify through programming the width of one or all column(s) of the object (list box, column or header) set using the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (a string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

Pass the new width (in pixels) of the object in the *width* parameter.

If *object* sets the list box object, all columns of the list box are resized.

If *object* sets a column or a column header, only the column set is resized.

The optional *minWidth* and *maxWidth* parameters can be used to set limits for the manual resizing of the column You can pass, respectively, the minimum and maximum width expressed in pixels in the *minWidth* and *maxWidth* parameters. If you want users to be unable to resize the column, you can pass the same value in *width*, *minWidth* and *maxWidth*.

# LISTBOX SET FOOTER CALCULATION

LISTBOX SET FOOTER CALCULATION ( {* ;} object ; calculation )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Variable (if * is omitted) |
| calculation | Longint | ⇒ | Calculation for footer area |

## Description

The **LISTBOX SET FOOTER CALCULATION** command sets the automatic calculation associated with the footer of the list box designated by the *object* and * parameters

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the object is a variable. In this case, you pass a variable reference instead of a string.
The *object* parameter can designate:

- the variable or name of a footer area. In this case, the command applies to this area.
- the variable or name of a list box column. In this case, the command applies to the footer area of this column.
- the variable or name of a list box. In this case, the command applies to all the footer areas of the list box.

In the *calculation* parameter, pass one of the following constants, found in the **Listbox Footer Calculation** theme, in order to set the calculation to be performed:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Listbox footer std deviation | Longint | 7 | Used with number or time type columns (only for array type list boxes)<br>Default type of the result: Real |
| lk footer average | Longint | 6 | Used with number or time type columns<br>Default type of the result: Real |
| lk footer count | Longint | 5 | Used with number, text, date, time, Boolean or picture type columns<br>Default type of the result: Longint |
| lk footer custom | Longint | 1 | No calculation performed by 4D. The footer variable must be calculated by programming.<br>Default type of the result: Footer variable type |
| lk footer max | Longint | 3 | Used with number, date, time or Boolean type columns<br>Default type of the result: Column array or field type |
| lk footer min | Longint | 2 | Used with number, date, time or Boolean type columns<br>Default type of the result: Column array or field type |
| lk footer sum | Longint | 4 | Used with number, time or Boolean type columns<br>Default type of the result: Column array or field type |
| lk footer sum squares | Longint | 9 | Used with number or time type columns (only for array type list boxes)<br>Default type of the result: Real |
| lk footer variance | Longint | 8 | Used with number or time type columns (only for array type list boxes)<br>Default type of the result: Real |

Note that predefined calculations take all the values of the column into account, including those of any hidden rows. If you want to restrict a calculation to only visible rows, you must use the lk footer custom constant and perform a customized calculation.

If the data type of a column or of even one column of the list box (when object designate a whole list box) is not compatible with the *calculation* set, the footer is not modified and the error 18 is generated. If a column contains a formula (selection type list box), the error 10 is generated.

**Note:** Footer area variables are typed automatically (when there are not typed through the code) with regards to the calculation set in the Property List (see **List box footer specific properties**). If the variable data type does not correspond to the result expected by the **LISTBOX SET FOOTER CALCULATION** command, a typing error is generated. For example, for a column displaying dates, if the footer calculation is 'Maximum', the *footer* variable will be typed as Date. At this point, if

you execute the statement **LISTBOX SET FOOTER CALCULATION**(footer;lk footer count), an error is generated because the expected data type of the result (longint) differs from the actual variable data type.

# LISTBOX SET FOOTERS HEIGHT

LISTBOX SET FOOTERS HEIGHT ( {* ;} object ; height {; unit} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| height | Longint | ⇒ | Row height |
| unit | Longint | ⇒ | Unit of height value:<br>0 or omitted = pixels, 1 = lines |

## Description

The **LISTBOX SET FOOTERS HEIGHT** command modifies by programming the height of the footer row in the list box designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string. You can designate either the list box or any footer of the list box.

Pass the height to set in the *height* parameter. By default, if you omit the *unit* parameter, this height is expressed in pixels. To change the unit, you can pass one of the following constants (found in the **List Box** theme), in the *unit* parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk lines | Longint | 1 | Height designates a number of lines. 4D calculates the height of a line according to the font |
| lk pixels | Longint | 0 | Height is a number of pixels (default). |

**Note:** For more information about the calculation of row heights, refer to the *Design Reference* manual.

# LISTBOX SET GRID

LISTBOX SET GRID ( {* ;} object ; horizontal ; vertical )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| horizontal | Boolean | ⇒ | True = show, False = hide |
| vertical | Boolean | ⇒ | True = show, False = hide |

## Description

The **LISTBOX SET GRID** command allows you to display or hide the horizontal and/or vertical grid lines that make up the grid in the list box object set using the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

Pass the Boolean values in *horizontal* and *vertical* that indicate if the corresponding grid lines should be displayed (**True**) or hidden (**False**). The grid is displayed by default.

# LISTBOX SET GRID COLOR

LISTBOX SET GRID COLOR ( {* ;} object ; color ; horizontal ; vertical )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| color | Longint | ⇒ | RGB color value |
| horizontal | Boolean | ⇒ | Use color for horizontal grid lines |
| vertical | Boolean | ⇒ | Use color for vertical grid lines |

## Description

The **LISTBOX SET GRID COLOR** command allows you to modify the color of the grid in the list box object set using the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

Pass the RGB color value in *color*. For more information on RGB colors, refer to the description of the **OBJECT SET RGB COLORS** command.

The *horizontal* and *vertical* parameters allow you to set the grid lines to which you will apply a color:

- If you pass **True** in *horizontal*, the color will be applied to horizontal grid lines. If you pass **False**, their color is not changed.
- If you pass **True** in *vertical*, the color will be applied to vertical grid lines. If you pass **False**, their color is not changed.

# LISTBOX SET HEADERS HEIGHT

LISTBOX SET HEADERS HEIGHT ( {* ;} object ; height {; unit} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| height | Longint | ⇒ | Row height |
| unit | Longint | ⇒ | Unit of height value:<br>0 or omitted = pixels, 1 = lines |

## Description

The **LISTBOX SET HEADERS HEIGHT** command modifies by programming the height of the header row in the list box designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

You can designate either the list box or any header of the list box.

Pass the height to set in the *height* parameter. By default, if you omit the *unit* parameter, this height is expressed in pixels. To change the unit, you can pass one of the following constants (found in the **List Box** theme), in the *unit* parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk lines | Longint | 1 | Height designates a number of lines. 4D calculates the height of a line according to the font |
| lk pixels | Longint | 0 | Height is a number of pixels (default). |

Headers must respect the minimum height set by the system. This height is 24 pixels under Windows and 17 pixels under Mac OS. If you pass a lower value in the *height* parameter, the minimum height is applied.

**Note:** For more information about calculation row heights, refer to the *Design Reference* manual.

# LISTBOX SET HIERARCHY

LISTBOX SET HIERARCHY ( {* ;} object ; hierarchical {; hierarchy} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| hierarchical | Boolean | ⇒ | True = hierarchical list box False = non-hierarchical list box |
| hierarchy | Pointer array | ⇒ | Array of pointers |

## Description

The **LISTBOX SET HIERARCHY** command lets you configure the list box object designated by the *object* and * parameters in hierarchical or non-hierarchical mode.

**Note:** This command only functions with list boxes based on arrays. When this command is used with a list box based on selections, it does nothing.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string.

The Boolean *hierarchical* parameter lets you specify the list box mode:

- if you pass True, the list box is displayed in hierarchical mode,
- if you pass False, the list box is displayed in non-hierarchical mode (standard array mode).

When you pass a list box in hierarchical mode, certain properties are automatically restricted. For more information, please refer to the **Managing Hierarchical List Boxes** section.

The *hierarchy* parameter is used to designate the arrays of the list box to be used to construct the hierarchy (see example). If you display the list box in hierarchical mode and omit this parameter:

- if the list box is already in hierarchical mode, the command does nothing.
- if the list box is in non-hierarchical mode and has never been declared hierarchical, the first array is used as the hierarchy by default.
- if the list box is in non-hierarchical mode but has previously been declared hierarchical, the last hierarchy is re-established.

## Example

Definition of the aCountry, aRegion and aCity arrays as the hierarchy of a list box:

```
ARRAY POINTER($ArrHierarch;3)
$ArrHierarch{1}:=->aCountry `First break level
$ArrHierarch{2}:=->aRegion `Second break level
$ArrHierarch{3}:=->aCity `Third break level
LISTBOX SET HIERARCHY(*;"mylistbox";True;$ArrHierarch)
```

# LISTBOX SET LOCKED COLUMNS

LISTBOX SET LOCKED COLUMNS ( {* ;} object ; numColumns )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| numColumns | Longint | ⇒ | Number of columns to lock |

## Description

The **LISTBOX SET LOCKED COLUMNS** command locks *numColumns* columns (starting from the first on the left) in the list box designated by the *object* and * parameters.

Locked columns are shown in the left part of the list box and do not scroll with the rest of the list box columns. For more information, refer to the *Design Reference* manual.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

In *numColumns*, you can pass any value included between 1 and the total number of columns in the list box minus 1. For a list box with X columns, if you pass a value > X-1 in *numColumns*, it will be reduced automatically to the value X-1.

To remove the column locking, pass 0 or a negative value in *numColumns*.

# LISTBOX SET ROW COLOR

LISTBOX SET ROW COLOR ( {* ;} object ; row ; color {; colorType} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or<br>Variable (if * is omitted) |
| row | Longint | ➡ | Row number |
| color | Longint | ➡ | RGB color value |
| colorType | Longint | ➡ | Listbox font color (default) or Listbox background color |

## Description

**Note:** This command only works with array type list boxes.

The **LISTBOX SET ROW COLOR** command sets the color for a row or a cell in the array list box designated by the *object* and * parameters.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the object parameter is a variable. In this case, you pass a variable reference instead of a string.

You can designate a list box or a list box column in the *object* parameter:

- When *object* designates a list box, the command applies to the row.
- When *object* designates a list box column, the command applies to the cell located at the column/row intersection.

In *row*, pass the number of the row where you want to apply the new color.

**Note:** The command does not take any hidden/shown states of the list box rows into account.

In *color*, pass an RGB color value. For more information about RGB colors, refer to the description of the **OBJECT SET RGB COLORS** command. If you want the row to inherit the color set for the higher level, you can pass the lk inherited constant in *color*.

In the *colorType* parameter, pass the lk background color or lk font color constants to indicate whether you want to apply the color as the background or the font color of the row. If you omit this parameter, the color is applied as the font color.

This command modifies values found in the array of colors that may have been defined for the column or list box. If these arrays are not already defined, the command dynamically creates arrays that you can access using the **LISTBOX Get array** command.

If conflicting color values are set using properties for list boxes or list box columns, an order of priority is applied. For more information, refer to the *Design Reference* manual.

## Example

In an array type list box, we want to set colors for a row and for one cell in this row:

```
 // Definition of font color for cell (yellow)
LISTBOX SET ROW COLOR(*;"Col5";3;0x00FFFF00)

 // Definition of background and font color for row 3
 // red background, blue font
LISTBOX SET ROW COLOR(*;"ListBox";3;0x00FF0000;lk background color)
LISTBOX SET ROW COLOR(*;"List Box";3;0x000000FF)
```

| | | | | | |
|---|---|---|---|---|---|
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |

# LISTBOX SET ROW FONT STYLE

LISTBOX SET ROW FONT STYLE ( {* ;} object ; row ; style )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or<br>Variable (if * is omitted) |
| row | Longint | ➡ | Row number |
| style | Longint | ➡ | Font style |

## Description

**Note:** This command only works with array type list boxes.

The **LISTBOX SET ROW FONT STYLE** command sets a font style for a row or a cell in the array type list box designated by the *object* and * parameters.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a is a variable. In this case, you pass a variable reference instead of a string.

You can designate a list box or a list box column in the *object* parameter:

- When *object* designates a list box, the command applies to the row.
- When *object* designates a list box column, the command applies to the cell located at the column/row intersection.

In *row*, pass the number of the row where you want to apply the new style.

**Note:** The command does not take any hidden/shown states of the list box rows into account.

In *style*, you pass a style value. You must use one (or a combination) of the constants found in the **Font Styles** theme:

| Constant | Type | Value |
|---|---|---|
| Bold | Longint | 1 |
| Italic | Longint | 2 |
| Plain | Longint | 0 |
| Underline | Longint | 4 |

If an array of font styles has been associated with the list box or column, only the element matching the row is modified. In other words, executing the command has the same effect, in this case, as modifying an element of the font style array.
If there is no font style array associated with the list box or column, one will be created dynamically when this command is called. You can access them using the **LISTBOX Get array** command.

If conflicting style properties are set for the column or the list box, an order of priority is applied. For more information, refer to the *Design Reference* manual.

**Note:** Since style arrays for columns take priority over the ones for list boxes, when this command is applied to a list box, it will only have an effect if no style array has been assigned to the columns.

## Example

Given an array type list box with the following characteristics:

- a font style array associated with the list box (*ArrGlobalStyle*)
- a font style array associated with column 5 (*ArrCol5Style*)
- the other columns do not have any style arrays.

```
LISTBOX SET ROW FONT STYLE(*;"Col5";3;Bold)
 // equivalent to ArrCol5Style{3}:=Bold
```

| | | | | | |
|---|---|---|---|---|---|
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | **text** | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |

```
LISTBOX SET ROW FONT STYLE(*;"List Box";3;Italic+Underline)
 // equivalent toArrGlobalStyle{3}:=Italic+Underline
```

| | | | | | |
|---|---|---|---|---|---|
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| *text* | *text* | *text* | *text* | **text** | *text* |
| text | text | text | text | text | text |
| text | text | text | text | text | text |
| text | text | text | text | text | text |

After the second statement, all the cells of the third row change to underlined italic, except for the one in the 5th column which stays in bold only (column style arrays take priority over list box arrays).

# LISTBOX SET ROW HEIGHT

LISTBOX SET ROW HEIGHT ( {* ;} object ; row ; height )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or Variable (if * is omitted) |
| row | Longint | ➡ | List box row whose height you want to set |
| height | Longint | ➡ | Height of list box row |

## 4D View Pro

This command requires a 4D View Pro license. If this license is not available, an error is displayed in the list box when the form is executed. For more information, please refer to the **4D View Pro** section.

## Description

The **LISTBOX SET ROW HEIGHT** command allows you to modify the height of the specified *row* in the list box object designated using the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

If the specified *row* does not exist in the list box, the command does nothing.

The unit used for the specified *height* corresponds to the one defined globally for the list box rows, either in the Property list or by a prior call to the **LISTBOX SET ROWS HEIGHT** command.

The **LISTBOX SET ROW HEIGHT** command modifies the row height array specified in the Property List, if any (for more information, please see the **Row Height Array** section in the *Design Reference* manual); otherwise, its creates a row height array dynamically. Using this command to set individual row heights produces the exact same visual result as associating a row height array using the Property List; however, filling a row height array with values is much faster than calling this command in a loop to set row heights one by one for the list box.

**Important note:** If the global **LISTBOX SET ROWS HEIGHT** command is called subsequently with a different unit than the one previously defined, the default value set by this command will replace and reinitialize any row heights set using **LISTBOX SET ROW HEIGHT** (see example 2).

## Example 1

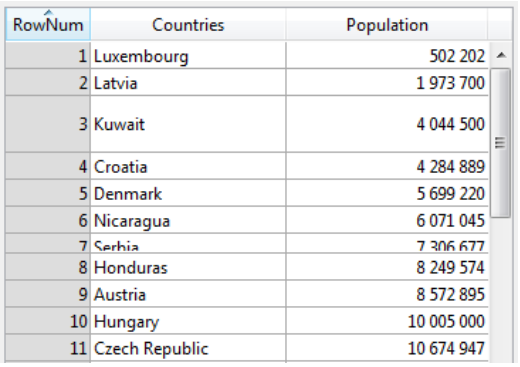
You want to change the height of a few rows in the following list box:

| RowNum | Countries | Population |
|---|---|---|
| 1 | Luxembourg | 502 202 |
| 2 | Latvia | 1 973 700 |
| 3 | Kuwait | 4 044 500 |
| 4 | Croatia | 4 284 889 |
| 5 | Denmark | 5 699 220 |
| 6 | Nicaragua | 6 071 045 |
| 7 | Serbia | 7 306 677 |
| 8 | Honduras | 8 249 574 |
| 9 | Austria | 8 572 895 |
| 10 | Hungary | 10 005 000 |
| 11 | Czech Republic | 10 674 947 |

If you execute this code:

```
 //current unit is pixels
LISTBOX SET ROW HEIGHT(*;"listboxname";3;40) //Kuwait
LISTBOX SET ROW HEIGHT(*;"listboxname";7;14) //Serbia
```

... you get the following result:



| RowNum | Countries | Population |
|---|---|---|
| 1 | Luxembourg | 502 202 |
| 2 | Latvia | 1 973 700 |
| 3 | Kuwait | 4 044 500 |
| 4 | Croatia | 4 284 889 |
| 5 | Denmark | 5 699 220 |
| 6 | Nicaragua | 6 071 045 |
| 7 | Serbia | 7 306 677 |
| 8 | Honduras | 8 249 574 |
| 9 | Austria | 8 572 895 |
| 10 | Hungary | 10 005 000 |
| 11 | Czech Republic | 10 674 947 |

## Example 2

You have set a default row height and then set several individual row height values using the **LISTBOX SET ROW HEIGHT** command:

```
LISTBOX SET ROWS HEIGHT(*;"listboxname";25;lk pixels) // global height set in pixels

LISTBOX SET ROW HEIGHT(*;"listboxname";1;30) // row 1: 30 pixels
LISTBOX SET ROW HEIGHT(*;"listboxname";5;40) // row 5: 40 pixels
LISTBOX SET ROW HEIGHT(*;"listboxname";11;50) // row 11: 50 pixels
```

Later, if the following code is executed…

```
LISTBOX SET ROWS HEIGHT(*;"listboxname";18;lk pixels)
```

…then the global row height is set to 18 pixels; however, since the unit has not changed, rows 1, 5 and 11 will keep their specific height values, i.e., 30, 40 and 50 pixels as defined above by the **LISTBOX SET ROW HEIGHT** command.

On the other hand, if the code below is executed subsequently…

```
LISTBOX SET ROWS HEIGHT(*;"listboxname";2;lk lines)
```

…then rows 1, 5 and 11 are reset to the global default row height set by **LISTBOX SET ROWS HEIGHT** (i.e., 2 lines) because the unit has changed from pixels to lines**.** Since there is no automatic conversion applied, changing units always results in row heights being reinitialized to the new default value defined.

# LISTBOX SET ROWS HEIGHT

LISTBOX SET ROWS HEIGHT ( {* ;} object ; height {; unit} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| height | Longint | ⇒ | Row height (in pixels) |
| unit | Longint | ⇒ | Unit of height value:<br>0 or omitted = pixels, 1 = lines |

## Description

The **LISTBOX SET ROWS HEIGHT** command allows you to modify by programming the row height in the list box object set using the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

By default, if you omit the *unit* parameter, the height is expressed in pixels. To modify the unit, in the *unit* parameter you can pass one of the following constants, found in the **List Box** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk lines | Longint | 1 | Height designates a number of lines. 4D calculates the height of a line according to the font |
| lk pixels | Longint | 0 | Height is a number of pixels (default). |

**Note:** For more information about calculating the height of rows, refer to the *Design Reference* manual.

# LISTBOX SET STATIC COLUMNS

LISTBOX SET STATIC COLUMNS ( {* ;} object ; numColumn )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| numColumn | Longint | ⇒ | Number of columns to make static |

## Description

The **LISTBOX SET STATIC COLUMNS** command sets *numColumns* columns as static (starting from the first on the left) in the list box designated by the *object* and * parameters.

Static columns cannot be moved within the list box.

**Note:** Static columns and locked columns are two independent functions. For more information, refer to the *Design Reference* manual.

# LISTBOX SET TABLE SOURCE

LISTBOX SET TABLE SOURCE ( {* ;} object ; tableNum | selName {; highlightName} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| tableNum \| selName | Longint, String | ⇒ | Number of table whose current selection is to be used or Named selection to be used |
| highlightName | String | ⇒ | Name of highlight set |

## Description

The **LISTBOX SET TABLE SOURCE** command can be used to modify the source of the data displayed in the listbox that is designated by the * and *object* parameters.

**Note:** This command can only be used when the “Data Source” property of the list box is set to **Current Selection** or **Named Selection** (for more information about this, please refer to the **Managing List Box Objects** section). It does nothing if you use it with a listbox that is associated with an array.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, please refer to the **Object Properties** section.

If you pass a table number as the *tableNum* parameter, the listbox will be filled in with the data of the records in the current selection of the table.

If you pass a named selection as the *selName* parameter, the listbox will be filled in with the data of the records belonging to the named selection.

The optional *highlightName* parameter associates a highlight set with the list box. The highlight set manages record highlighting by the user in the list box.

If the listbox already contains columns, their contents will be updated after the command is executed.

**Note:** For optimization purposes, this command is processed in an asynchronous manner; in other words, the source of the listbox is changed only after the complete execution of the method in which the command is called.

# LISTBOX SORT COLUMNS

LISTBOX SORT COLUMNS ( {* ;} object ; colNum ; order {; colNum2 ; order2 ; ... ; colNumN ; orderN} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| colNum | Longint | ⇒ | Column number(s) to sort |
| order | Operator | ⇒ | ">" to sort in ascending order or "<" to sort in descending order |

## Description

The **LISTBOX SORT COLUMNS** command sorts the rows of the list box set in the *object* and * parameters on the basis of one or more column value(s).

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string. For more information about object names, refer to the **Object Properties** section.

In *colNum*, pass the column number of the column whose values you want to use as the sort criteria. You can use any type of array data, except pictures and pointers.

In *order*, pass the symbol > or < to indicate the sort order. If *order* contains the “greater than” symbol (>), the sort order is ascending. If *order* contains the “less than” symbol (<), the sort oder is descending.

You can define multi-level sorts: to do so, pass as many pairs (*colNum;order*) as necessary. The sorting level is defined by the position of the parameter in the call.

In conformity with the principle of list box operation, the columns are synchronized which means that the sorting of a column is automatically passed on to all the other columns of the object.

# Math

- Abs
- Arctan
- Cos
- Dec
- Euro converter
- Exp
- Int
- Log
- Mod
- Random
- Round
- SET REAL COMPARISON LEVEL
- Sin
- Square root
- Tan
- Trunc

# Abs

Abs ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number for which to return the absolute value |
| Function result | Real | ⇐ | Absolute value of number |

## Description

**Abs** returns the absolute (unsigned, positive) value of *number*. If *number* is negative, it is returned as positive. If *number* is positive, it is returned unchanged.

## Example

The following example returns the absolute value of −10.3, which is 10.3:

```
vlVector:=Abs(-10.3)
```

# Arctan

Arctan ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Tangent for which to calculate the angle |
| Function result | Real | ⇒ | Angle in radians |

## Description

**Arctan** returns the angle, expressed in radians, of the tangent *number*.

**Note:** 4D provides the predefined constants Pi, Degree, and Radian. Pi returns the Pi number (3.14159...), Degree returns one degree expressed in radians (0.01745...), and Radian returns one radian expressed in degrees (57.29577...).

## Example

The following example shows the value of Pi:

```
ALERT("Pi is equal to: "+String(Arctan(1)*4))
```

# Cos

Cos ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number, in radians, whose cosine is returned |
| Function result | Real | ⇐ | Cosine of number |

## Description

**Cos** returns the cosine of *number*, where *number* is expressed in radians.

**Note:** 4D provides the predefined constants Pi, Degree, and Radian. Pi returns the Pi number (3.14159...), Degree returns one degree expressed in radians (0.01745...), and Radian returns one radian expressed in degrees (57.29577...).

# Dec

Dec ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number whose decimal portion is returned |
| Function result | Real | ⮌ | Decimal part of number |

## Description

**Dec** returns the decimal (fractional) portion of *number*.The value returned is always positive or zero.

## Example

The following example takes a monetary value expressed as a real number, and extracts the dollar part and the cents part. If *vrAmount* is 7.31, then *vlDollars* is set to 7 and *vlCents* is set to 31:

```
vlDollars:=Int(vrAmount) ` Get the dollars
vlCents:=Dec(vrAmount)*100 ` Get the fractional part
```

# Euro converter

Euro converter ( value ; fromCurrency ; toCurrency ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| value | Real | ⇒ | Value to convert |
| fromCurrency | String | ⇒ | Code of the currency in which the value is expressed |
| toCurrency | String | ⇒ | Code of the currency into which the value must be converted |
| Function result | Real | ⇐ | Converted value |

## Description

The **Euro converter** command converts any value from and to the different currencies belonging to “Euroland” and the Euro currency itself.

You can convert:

- a national currency into Euros,
- Euros into a national currency,
- a national currency into another national currency. In this case, the conversion is calculated by the intermediary of the Euro, as specified in the European reglementation. For example, to convert Belgian francs to Deutschemarks, 4D will perform the following calculations: Belgian francs -> Euros -> Deutchemarks.

Pass the value to convert in the first parameter.
The second parameter indicates the Currency code in which value is expressed.
The third parameter indicates the Currency code into which value must be converted.

To specify a Currency code, 4D proposes the following predefined constants, placed in the “**Euro Currencies**” theme:

| Constant | Type | Value |
|---|---|---|
| Austrian Schilling | String | ATS |
| Belgian Franc | String | BEF |
| Deutsche Mark | String | DEM |
| Euro | String | EUR |
| Finnish Markka | String | FIM |
| French Franc | String | FRF |
| Greek Drachma | String | GRD |
| Irish Pound | String | IEP |
| Italian Lira | String | ITL |
| Luxembourg Franc | String | LUF |
| Netherlands Guilder | String | NLG |
| Portuguese Escudo | String | PTE |
| Spanish Peseta | String | ESP |

If necessary, 4D performs rounding automatically on conversion results and keeps 2 decimals —except for conversions to Italian Lires, Belgian Francs, Luxembourg Francs and Spanish Pesetas, for which 4D keeps 0 decimal (the result is an integer number).

The conversion rates between the Euro and the currencies of the 11 participating Member States are fixed:

| Currency | Value for 1 Euro |
|---|---|
| Austrian Schilling | 13.7603 |
| Belgian Franc | 40.3399 |
| Deutschemark | 1.95583 |
| Finnish Markka | 5.94573 |
| French Franc | 6.55957 |
| Greek drachma | 340.750 |
| Irish Pound | 0.787564 |
| Italian Lire | 1936.27 |
| Luxembourg Franc | 40.3399 |
| Netherlands Guilder | 2.20371 |
| Portuguese Escudo | 200.482 |
| Spanish Peseta | 166.386 |

## Example

Here are some examples of conversions that can be done with this command:

```
$value:=10000 `Value expressed in French Francs
 `Convert the value into Euros
$InEuros:=Euro converter($value;French Franc;Euro)
 `Convert the value into Italian Lire
$InLires:=Euro converter($value;French Franc;Italian Lire)
```

# Exp

Exp ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number to evaluate |
| Function result | Real | ⇒ | Natural log base by the power of number |

## Description

**Exp** raises the natural log base (e = 2.71828...) by the power of *number*. **Exp** is the inverse function of **Log**.

**Note:** 4D provides the predefined constant *e number* (2.71828...).

## Example

The following example assigns the exponential of 1 to *vrE* (the log of *vrE* is 1):

```
vrE:=Exp(1) ` vrE gets 2.17828...
```

# Int

Int ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number whose integer portion is returned |
| Function result | Real | ⮌ | Integer portion of number |

## Description

**Int** returns the integer portion of *number*. **Int** truncates a negative *number* away from zero.

## Example

The following example illustrates how **Int** works for both positive and negative numbers. Note that the decimal portion of the number is removed:

```
vlResult:=Int(123.4) ` vlResult gets 123
vlResult:=Int(-123.4) ` vlResult gets -124
```

# Log

Log ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number for which to return the log |
| Function result | Real | ⇒ | Log of number |

## Description

**Log** returns the natural (Napierian) log of *number*. **Log** is the inverse function of **Exp**.

**Note:** 4D provides the predefined constant *e number* (2.71828...).

## Example

The following line displays 1:

```
ALERT(String(Log(Exp(1))))
```

# Mod

Mod ( number1 ; number2 ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number1 | Longint | ⇨ | Number to divide |
| number2 | Longint | ⇨ | Number to divide by |
| Function result | Real | ⇦ | Returns the remainder |

## Description

The **Mod** command returns the remainder of the Integer division of *number1* by *number2*.

**Notes:**

- **Mod** accepts Integer, Long Integer, and Real expressions. However, if *number1* or *number2* are real numbers, the numbers are first rounded and then **Mod** is calculated.
- Be careful when using **Mod** with real numbers of a large size (above 2^31) since, in this case, its operation may reach the limits of the calculation capacities of standard processors.

You can also use the % operator to calculate the remainder (see **Numeric Operators**).

**WARNING:** The % operator returns valid results with Integer and Long Integer expressions. To calculate the modulo of real values, you must use the **Mod** command.

## Example

The following example illustrates how the **Mod** function works with different arguments. Each line assigns a number to the *vlResult* variable. The comments describe the results:

```
vlResult:=Mod(3;2) ` vlResult gets 1
vlResult:=Mod(4;2) ` vlResult gets 0
vlResult:=Mod(3.5;2) ` vlResult gets 0
```

# Random

Random -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ⮌ | Random number |

## Description

**Random** returns a random integer value between 0 and 32,767 (inclusive).

To define a range of integers from which the random value will be chosen, use this formula:

```
(Random%(vEnd-vStart+1))+vStart
```

The value *vStart* is the first number in the range, and the value *vEnd* is the last.

## Example

The following example assigns a random integer between 10 and 30 to the *vlResult* variable:

```
vlResult:=(Random%21)+10
```

# Round

Round ( round ; places ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| round | Real | ⇒ | Number to be rounded |
| places | Longint | ⇒ | Number of decimal places used for rounding |
| Function result | Real | ⇐ | Number rounded to the number of decimal places specified by Places |

## Description

**Round** returns *number* rounded to the number of decimal places specified by *places*.

If *places* is positive, *number* is rounded to *places* decimal places. If *places* is negative, *number* is rounded on the left of the decimal point.

If the digit following *places* is 5 though 9, **Round** rounds toward positive infinity for a positive number, and toward negative infinity for a negative number. If the digit following *places* is 0 through 4, **Round** rounds toward zero.

## Example

The following example illustrates how **Round** works with different arguments. Each line assigns a number to the *vlResult* variable. The comments describe the results:

```
vlResult:=Round(16.857;2) ` vlResult gets 16.86
vlResult:=Round(32345.67;-3) ` vlResult gets 32000
vlResult:=Round(29.8725;3) ` vlResult gets 29.873
vlResult:=Round(-1.5;0) ` vlResult gets -2
```

# SET REAL COMPARISON LEVEL

SET REAL COMPARISON LEVEL ( epsilon )

| Parameter | Type | | Description |
|---|---|---|---|
| epsilon | Real | ⇒ | Epsilon value for real equality comparisons |

## Description

The **SET REAL COMPARISON LEVEL** command sets the epsilon value used by 4D to compare real values and expressions for equality.

A computer always performs approximative real computations; therefore, testing real numbers for equality should take this approximation into account. 4D does this when comparing real numbers by testing whether or not the difference between the two numbers exceeds a certain value. This value is called the **epsilon** and works this way:

Given two real numbers *a* and *b*, if **Abs***(a-b)* is greater than the epsilon, the numbers are considered not equal; otherwise, the numbers are considered equal.

By default, 4D, sets the epsilon value to 10 power minus 6 (10^-6). Please note that the *epsilon* value should always be positive. Examples:

- *0.00001=0.00002* returns False, because the difference *0.00001* is greater than *10^-6*.
- *0.000001=0.000002* returns True, because the difference *0.000001* is not greater than *10^-6*.
- *0.000001=0.000003* returns False, because the difference *0.000002* is greater than *10^-6*.

Using **SET REAL COMPARISON LEVEL**, you can increase or decrease the epsilon value as you require.

**WARNING:** Typically, you will not need to use this command to change the default epsilon value.

**IMPORTANT:** Changing the epsilon only affects real comparison for equality. It has no effect on other real computations nor on the display of real values.

**Note:** The **SET REAL COMPARISON LEVEL** command has no effect on queries and sorts performed with fields of the Real type. It only applies the 4D language.

# Sin

Sin ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number, in radians, whose sine is returned |
| Function result | Real | ⇐ | Sine of number |

## Description

**Sin** returns the sine of *number*, where *number* is expressed in radians.

**Note:** 4D provides the predefined constants Pi, Degree, and Radian. Pi returns the Pi number (3.14159...), Degree returns one degree expressed in radians (0.01745...), and Radian returns one radian expressed in degrees (57.29577...).

# Square root

Square root ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number whose square root is calculated |
| Function result | Real | ⇒ | Square root of the number |

## Description

**Square root** returns the square root of *number*.

## Example 1

The line:

```
$vrSquareRootOfTwo :=Square root(2)
```

assigns the value *1.414213562373* to the variable *$vrSquareRootOfTwo*.

## Example 2

The following method returns the hypotenuse of the right triangle whose two legs are passed as parameters:

```
  ` Hypotenuse method
  ` Hypotenuse ( real ; real ) -> real
  ` Hypotenuse ( legA ; legB ) -> Hypotenuse
C_REAL($0;$1;$2)
$0:=Square root(($1^2)+($2^2))
```

For instance, Hypotenuse (4;3) returns 5.

# Tan

Tan ( number ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number, in radians, whose tangent is returned |
| Function result | Real | ⇐ | Tangent of number |

## Description

**Tan** returns the tangent of *number*, where *number* is expressed in radians.

**Note:** 4D provides the predefined constants Pi, Degree, and Radian. Pi returns the Pi number (3.14159...), Degree returns one degree expressed in radians (0.01745...), and Radian returns one radian expressed in degrees (57.29577...).

# Trunc

Trunc ( number ; places ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| number | Real | ⇒ | Number to be truncated |
| places | Longint | ⇒ | Number of decimal places used for truncating |
| Function result | Real | ⇐ | Number with its decimal part truncated to the number of decimal places specified by Places |

## Description

**Trunc** returns *number* with its decimal part truncated to the number of decimal places specified by *places*. **Trunc** always truncates toward negative infinity.

If *places* is positive, *number* is truncated to *places* decimal places. If *places* is negative, number is truncated on the left of the decimal point.

## Example

The following example illustrates how **Trunc** works with different arguments. Each line assigns a number to the *vlResult* variable. The comments describe the results:

```
vlResult:=Trunc(216.897;1) ` vlResult gets 216.8
vlResult:=Trunc(216.897;-1) ` vlResult gets 210
vlResult:=Trunc(-216.897;1) ` vlResult gets -216.9
vlResult:=Trunc(-216.897;-1) ` vlResult gets -220
```

# Menus

- Managing Menus
- APPEND MENU ITEM
- Count menu items
- Count menus
- Create menu
- DELETE MENU ITEM
- DISABLE MENU ITEM
- ENABLE MENU ITEM
- Get menu bar reference
- Get menu item
- GET MENU ITEM ICON
- Get menu item key
- Get menu item mark
- Get menu item method
- Get menu item modifiers
- Get menu item parameter
- GET MENU ITEM PROPERTY
- Get menu item style
- GET MENU ITEMS
- Get menu title
- Get selected menu item parameter
- INSERT MENU ITEM
- Menu selected
- RELEASE MENU
- SET MENU BAR
- SET MENU ITEM
- SET MENU ITEM ICON
- SET MENU ITEM MARK
- SET MENU ITEM METHOD
- SET MENU ITEM PARAMETER
- SET MENU ITEM PROPERTY
- SET MENU ITEM SHORTCUT
- SET MENU ITEM STYLE

# Managing Menus

**Terminology:** The documentation of Menus commands uses the terms **menu command** and **menu item** interchangeably when describing a line in a menu.

## MenuRef and Menu Numbers

The language of 4D offers two ways of managing menus and menu bars: by references or by numbers.

- Managing menus by reference (MenuRef) is the new way of handling menus, introduced with version 11 of 4D. This mode gives access to advanced functions such as the creation of completely dynamic interfaces (menus created "on the fly" without necessarily existing in the Menu editor) and the managing of multi-level hierarchical submenus.
- Managing menus and menu bars by number is based on the menus created in the Menu editor in Design mode. Each menu bar and menu is assigned a fixed number (corresponding to its position in the editor). This number is used by the language commands to specify the menu bar or menu. The scope of language commands applied to menus managed by number is the current menu bar.
  This behavior corresponds to previous versions of 4D and complies with several rules (described below in the "Managing menus by number" paragraph). It can still be used but does not take advantage of the new functions offered starting with version 11, more particularly the dynamic management of menus and the use of hierarchical submenus: it is not possible to access a hiearchical submenu using a number.

Both menu management modes are compatible and can be used simultaneously in your interfaces. Most of the commands in the “Menus” theme accept both menu numbers and references indiscriminately.

However, managing menus by reference is recommended since it offers many more possibilities. Note that if your menu interface is partially or completely defined via the Menu editor, it remains entirely possible to work with it in the form of references using the **Get menu bar reference** and **GET MENU ITEMS** commands.

## Managing menus by reference

When menus are handled by means of MenuRef references, there is no difference per se between a menu and a menu bar. In both cases, it consists of a list of items. Only their use differs. In the case of a menu bar, each item corresponds to a menu which is itself composed of items. This is also the principle on which hierarchical menus are based: each item can itself be a menu, and so on.

When a menu is managed by reference, any changes made to this menu during the session are immediately passed on to every instance of this menu and in every process of the database.

### MenuRef

Like hierarchical lists, every menu has a unique reference, with which it can be identified during the entire session. This reference, named by convention *MenuRef*, is a 16-character alphanumeric. All the commands of the “Menus” theme accept either this reference, or a menu number, to specify a menu or menu bar.

Menu references can be obtained using the **Create menu**, **Get menu bar reference** or **GET MENU ITEMS** commands.

## Managing menus by number

### Menu Bars

Menu bars can be defined in the Menu editor in Design mode. When managed by number, each menu bar is identified by a number and by a name. The first menu bar (created automatically by 4D) has the number 1 and is named Menu Bar #1 by default. You can rename it in the Menu editor. The name of a menu bar may contain up to 31 characters and must be unique.

Menu Bar #1 is also the default menu bar. To open an application with a menu bar other than Menu Bar #1, you must use the **SET MENU BAR** command in the **On Startup database method**. It is not possible to modify the contents of a menu bar itself by programming; however, the menus comprising it can be modified. The scope of the language commands applied to

static menus is the current menu bar. On each call to the **SET MENU BAR** command (without the * parameter), all the menus and menu commands return to their original state as defined in the Menu editor.

Every menu bar comes equipped with three menus—the **File**, **Edit** and **Mode** menus.

- The **File** menu has only one menu command—Quit. The Quit standard action is assigned to it. This action displays an "Are you sure?" confirmation dialog box then quits the 4D application if this dialog box is validated. Otherwise, the operation is cancelled.

**Note:** Under Mac OS X, the created menu command associated with the Quit action is automatically placed in the application menu when the database is executed on this system.

You can rename the File menu, add menu commands to it or keep it as is. It is recommended that you always keep Quit as the last menu command in the File menu.

- The **Edit** menu contains the standard editing menu commands. A standard action (Cancel, Cut, Copy, etc.) is assigned to each command of this menu. You can add commands to this menu or use your own methods for managing editing actions.
- The **Mode** menu contains the Return to Design mode command. This command can be used to return to the Design mode (when it is available) from the Application mode.

**Note:** 4D automatically manages the **Help** and **application** (Mac OS X) system menus. These menus cannot be modified, except for the **About 4D** command, which can be managed using the **SET ABOUT** command.

**Warning:** Menu bars are "interprocess." Any modification carried out on a menu bar in the Design mode will be reflected in all the processes where the menu bar is used.

### Menu Numbers and Menu Command Numbers

Like menu bars, menus are numbered. The **File** menu is generally menu 1. Thereafter, menus are numbered sequentially from left to right (2, 3, 4, and so on). The **Application** menu (Mac OS ) is excluded from this numbering. On both platforms, the **Help** menu is also excluded. It should be noted that the **Count menus** command does not take these menus into account. If, for example, your menu bar consists of the File, Edit, Customers, Inoices and Help menus, **Count menus** will return 4 (ignoring the system menus maintained by 4D).

Menu numbering is important when you are working, for example, with the **Menu selected** function.

When a menu is associated with a form, the menu numbering scheme is different. The first appended menu begins with the number 2049. To refer to an appended menu, add 2048 to the normal menu number.

The menu commands within each menu are numbered sequentially from the top of the menu to the bottom including the separators. The topmost menu command is item 1.

### Associated Menu Bars

You can associate a menu bar with a form in the Form properties (General page). Such a menu bar is called a “form menu bar” in this document.

The menus on a form menu bar are appended to the current menu bar when the form is displayed as an output form in the Application environment.

Form menu bars are specified by a menu bar number and a name. If the number or name of the menu bar displayed with the current form is the same as that of the menu bar appended to the form, the menu bar is not appended.

By default, when a form is displayed with a custom menu bar, the commands of the current menu bar are deactivated, i.e. selecting them has no effect. You can modify this operation by checking the **Active Menu Bar** option in the Form properties: in this case, the commands of the current menu bar will remain usable.

In every case, the selection of a menu command causes an On Menu Selected event to be sent to the form method; you can then use the **Menu selected** command to test the selected menu.

### Attached Menus

Menus can be attached to menu bars. If an attached menu is modified using one of these commands, every other instance of the menu will reflect these changes. For more information about attaching menus, refer to the 4D Design Reference Manual.

## Standard actions and methods associated with menu commands

Each menu command can have a project method or a standard action attached to it. If you do not assign a method or a standard action to a menu command, choosing that menu command causes 4D to exit the Application environment and go to the Design environment. If only the Application environment is available or if the user does not have access to the Design environment, this means quitting to the Desktop.

Standard actions can be used to carry out various current operations linked to system functions (copy, quit, etc.) or to those of the 4D database (add record, select all, etc.).

You can assign both a standard action and a project method to a menu command. In this case, the standard action is never executed; however, 4D uses this action to activate/deactivate the menu command according to the current context and to associate a specific operation with it according to the platform (for example, the Preferences action is passed in the application menu under Mac OS). When a menu command is deactivated, the associated project method cannot be executed.

## menuItem=-1

In order to facilitate the managing of menu items, 4D provides a shortcut that can be used to specify the last item added to the menu: you simply need to pass -1 in the *menuItem* parameter.

This principle can be used in all the commands of the “Menus” theme that work with menu items.

# APPEND MENU ITEM

APPEND MENU ITEM ( menu ; itemText {; subMenu {; process {; *}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| itemText | Text | ⇒ | Text for the new menu items |
| subMenu | MenuRef | ⇒ | Reference of submenu associated with item |
| process | Longint | ⇒ | Process reference number |
| * | Operator | ⇒ | If passed: consider metacharacters as standard characters |

## Description

The **APPEND MENU ITEM** command appends new menu items to the menu whose number or reference is passed in *menu*.

If you omit the *process* parameter, **APPEND MENU ITEM** applies to the menu bar for the current process. Otherwise, **APPEND MENU ITEM** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purporse and will be ignored.

If you do not pass the * parameter, **APPEND MENU ITEM** allows you to append one or several menu items in one call.

You define the items to be appended with the parameter *itemText* as follows:

- Separate each item from the next one with a semi-colon (*;*). For example, *"ItemText1;ItemText2;ItemText3".*
- To disable an item: Place an opening parenthesis (*(*) in the item text.
- To specify a separation line: Pass "-" or "(-" as item text.
- To specify a font style for a line: In the item text, place a less than sign (<) followed by one of these characters:

| | |
|---|---|
| <B | Bold |
| <I | Italic |
| <U | Underline |

- To add a check mark to an item: In the item text, place an exclamation mark (*!*) followed by the character you want as a check mark. On Macintosh, the character is displayed; on Windows, a check mark is displayed no matter what character you passed.
- To add an icon to an item: In the item text, place a circumflex accent (^) followed by a character whose code plus 208 is the resource ID of a Mac OS-based icon resource.
- To add a shortcut to an item: In the item text, place a slash (*/*) followed by the shortcut character for the item.

**Note:** Use menus that have a reasonable number of items. For example, if you want to display more than 50 items, consider using a scrollable area in a form instead of a menu.

If you pass the * parameter, the "special" characters (*; ( !*...) included in the item text will be considered as standard characters and not as metacharacters. This means that you can create menu items like "**Copy (special)...**" or "**Find/Replace...**". Note that when the * parameter is passed, you cannot create several items in a single call since the ";" character is considered as a standard character.

**Note:** The **GET MENU ITEMS** and **Get menu item** commands will or will not return any metacharacters in the text of a menu item depending on how it was created: if it was created with the * option, metacharacters will be returned as standard characters.

The optional *subMenu* parameter can be used to indicate a menu as the added item and thus position a hierarchical submenu. You must pass a menu reference (*MenuRef* type string) specifying a menu created, for example, using the **Create menu** command. If the command adds more than one menu item, the submenu is associated with the first item.

**Important:** The new items do not have any associated methods or actions. These must be associated with the items using the **SET MENU ITEM PROPERTY** or **SET MENU ITEM METHOD** commands, or the items can also be managed from within a form method using the **Menu selected** command.

## Example

This example appends the names of the available fonts to the Font menu, which in this example is the sixth menu of the current menu bar:

```
  ` In the On Startup database method
  ` The font list is loaded and menu item text is built
FONT LIST(◊asAvailableFont)
◊atFontMenuItems:=""
For($vlFont;1;Size of array(◊asAvailableFont))
   ◊atFontMenuItems:=◊atFontMenuItems+";"+◊asAvailableFont{$vlFont}
End for
```

Then, in any form or project method, you can write:

```
APPEND MENU ITEM(6;◊atFontMenuItems)
```

# Count menu items

Count menu items ( menu {; process} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| process | Longint | ⇒ | Process reference number |
| Function result | Longint | ⇐ | Number of menu items in the menu |

## Description

The **Count menu items** command returns the number of menu items present in the menu whose number or reference is passed in *menu.*

If you omit the *process* parameter, **Count menu items** applies to the menu bar for the current process. Otherwise, **Count menu items** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* parameter in *menu*, the *process* parameter serves no purpose and will be ignored.

# Count menus

Count menus {( process )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ⇒ | Process reference number |
| Function result | Longint | ⇐ | Number of menus in the current menu bar |

## Description

The **Count menus** command returns the number of menus present in the menu bar.

If you omit the *process* parameter, **Count menus** applies to the menu bar for the current process. Otherwise, **Count menus** applies to the menu bar for the process whose reference number is passed in *process*.

# Create menu

Create menu {{ menu )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | MenuRef, Longint, String | ⇒ | Menu reference or Number or Name of menu bar |
| Function result | MenuRef | ⇐ | Menu reference |

## Description

The **Create menu** command creates a new menu in memory. This menu will only exist in memory and will not be added in the Menu editor in Design mode. Any changes made to this menu during the session will be immediately carried over to all the instances of this menu and in all the processes of the database.

The command returns an ID of the *MenuRef* type for the new menu.

- If you do not pass the optional *menu* parameter, the menu is created blank. You must build and manage it using the **RELEASE MENU**, **SET MENU ITEM**, etc. commands.
- If you pass the *menu* parameter, the menu created will be an exact copy of the source menu designated by this parameter. All the properties of the source menu, including any associated submenus, will be applied to the new menu. Note that a new *MenuRef* reference is created for the source menu and for any existing submenus that are associated with it.

In the *menu* parameter, you can pass either a valid menu reference, or the number or name of a menu bar defined in Design mode. In this last case, the new menu will be made up of the menus and submenus of the source menu bar.

**Note:** If you pass an invalid value in *menu*, a blank menu is created.

A menu created by this command can be used as the menu bar using the **SET MENU BAR** command.

When you no longer need the menu created by **Create menu**, remember to call the **RELEASE MENU** command in order to free up the memory being used.

## Example

Refer to the example of the **SET MENU BAR** command.

# DELETE MENU ITEM

DELETE MENU ITEM ( menu ; menuItem {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| menuItem | Longint | ⇒ | Menu item number or -1 for last item added |
| process | Longint | ⇒ | Process reference number |

## Description

The **DELETE MENU ITEM** command deletes the menu item whose menu number or reference is passed in *menu* and whose item number is passed in *menuItem*. You can pass -1 in *menuItem* in order to indicate the last item added to *menu*.

If the menu item specified by *menu* and *menuItem* is itself a menu managed by reference and created, for example, using the **Create menu** command, **DELETE MENU ITEM** will only delete the instance of the *menuItem* in *menu*. The submenu referenced by the *menuItem* will continue to exist in memory. You must use the **RELEASE MENU** command in order to definitively delete a menu that is managed by reference.

This command also works with a menu bar created using the **Create menu** command and installed with the **SET MENU BAR** command.

If you omit the *process* parameter, **DELETE MENU ITEM** applies to the menu bar for the current process. Otherwise, **DELETE MENU ITEM** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

**Note:** For consistency in the user interface, do not keep a menu with no items.

# DISABLE MENU ITEM

DISABLE MENU ITEM ( menu ; menuItem {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| menuItem | Longint | ⇒ | Menu item number or -1 for the last item added |
| process | Longint | ⇒ | Proces reference number |

## Description

The **DISABLE MENU ITEM** command disables the menu item whose menu number or reference is passed in *menu* and whose item number is passed in *menuItem*.You can pass -1 in *menuItem* in order to designate the last item added to the *menu*.

If you omit the *process* parameter, **DISABLE MENU ITEM** applies to the menu bar for the current process. Otherwise, **DISABLE MENU ITEM** applies to the menu bar for the process whose reference number is passed in *process*.

If the *menuItem* parameter designates a hierarchical submenu, all the items of this menu and any submenus are disabled. This command also works with a menu bar created using the **Create menu** command and installed with the **SET MENU BAR** command.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

**Tip:** To enable/disable all items of a menu at once, pass 0 (zero) in *menuItem*.

# ENABLE MENU ITEM

ENABLE MENU ITEM ( menu ; menuItem {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| menuItem | Longint | ⇒ | Menu item number or -1 for the last item added |
| process | Longint | ⇒ | Proces reference number |

## Description

The **ENABLE MENU ITEM** command enables the menu item whose menu number or reference is passed in *menu* and whose item number is passed in *menuItem*. You can pass -1 in *menuItem* in order to designate the last item added to the *menu*.

If the *menuItem* parameter designates a hierarchical submenu, all the items of this menu and any submenus are enabled. This command also works with a menu bar created using the **Create menu** command and installed with the **SET MENU BAR** command.

If you omit the *process* parameter, **ENABLE MENU ITEM** applies to the menu bar for the current process. Otherwise, **ENABLE MENU ITEM** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

**Tip:** To enable/disable all items of a menu at once, pass 0 (zero) in *menuItem*.

# Get menu bar reference

Get menu bar reference {( process )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ⇒ | Reference number of process |
| Function result | MenuRef | ⇄ | Menu bar ID |

## Description

The **Get menu bar reference** command returns the ID of the current menu bar or the menu bar of a specific process.

If the menu bar was created by the **Create menu** command, this ID corresponds to the reference ID of the menu created. Otherwise, the command returns a specific internal ID. In all cases, this *MenuRef* ID may be used to reference the menu bar by all the other commands of the theme.

The *process* parameter can be used to designate the process where you want to get the current menu bar ID. If you omit this parameter, the command returns the menu bar ID of the current process.

## Example

Refer to the example of the **GET MENU ITEMS** command.

# Get menu item

Get menu item ( menu ; menuItem {; process} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| menuItem | Longint | ⇒ | Menu item number or -1 for last item added |
| process | Longint | ⇒ | Process reference number |
| Function result | String | ⇐ | Text of the menu item |

## Description

The **Get menu item** command returns the text of the menu item whose menu and item numbers are passed in *menu* and *menuItem*. You can pass -1 in *menuItem* in order to indicate the last item added to *menu*.

If you omit the *process* parameter, **Get menu item** applies to the menu bar for the current process. Otherwise, **Get menu item** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

# GET MENU ITEM ICON

GET MENU ITEM ICON ( menu ; menuItem ; iconRef {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu reference or Menu number |
| menuItem | Longint | ⇒ | Number of menu item or -1 for the last item added to the menu |
| iconRef | Text variable, Longint variable | ⇐ | Name or number of picture associated with menu item |
| process | Longint | ⇒ | Process number |

## Description

The **GET MENU ITEM ICON** command returns, in the *iconRef* variable, the reference of any icon that is associated with the menu item designated by the *menu* and *menuItem* parameters. This reference is the name or number of the picture.

You can pass -1 in *menuItem* in order to specify the last item added to *menu*.

In *menu*, you can pass a menu reference (*MenuRef*) or a menu number. If you pass a menu reference, the *process* parameter is unnecessary and will be ignored if it is passed. If you pass a menu number, the command will take the corresponding menu in the main menu bar of the current process into account. If you want to designate another process, pass its number in the optional *process* parameter.

If the icon has been specified using a library picture, the command returns either the name or number of the picture depending on the type of variable passed in this parameter. If the icon has been specified using a picture stored in the **Resources** folder of the database, the command returns the picture pathname in *iconRef*.

If you do not attribute a specific type to the *iconRef* variable, by default, the name of the picture is returned (text type).

If no icon is associated with the menu item, the command returns a blank value.

# Get menu item key

Get menu item key ( menu ; menuItem {; process} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| menuItem | Longint | ⇒ | Menu item number or -1 for the last item added |
| process | Longint | ⇒ | Process reference number |
| Function result | Longint | ⇐ | Character code of standard shortcut key associated with the menu item |

## Description

The **Get menu item key** command returns the code of the **Ctrl** (Windows) or **Command** (Macintosh) shortcut for the menu item whose menu number or reference is passed in *menu* and whose item number is passed in *menuItem*. You can pass -1 in *menuItem* in order to indicate the last item added to *menu*.

If you omit the *process* parameter, **Get menu item key** applies to the menu bar for the current process. Otherwise, **Get menu item key** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

If the menu item has no associated shortcut or if the *menuItem* parameter designates a hierarchical submenu, **Get menu item key** returns *0* (zero).

## Example

To obtain the shortcut associated with a menu item, it is useful to implement a programming structure of the following type:

```
If(Get menu item key(mymenu;1)#0)
   $modifiers:=Get menu item modifiers(mymenu;1)
   Case of
      :($modifiers=Option key mask)
         ...
      :($modifiers=Shift key mask)
         ...
      :($modifiers=Option key mask+Shift key mask)
         ...
   End case
End if
```

# Get menu item mark

Get menu item mark ( menu ; menuItem {; process} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ➡ | Menu number or Menu reference |
| menuItem | Longint | ➡ | Menu item number or -1 for last item added |
| process | Longint | ➡ | Process reference number |
| Function result | String | ⮌ | Current menu item mark |

## Description

The **Get menu item mark** command returns the check mark of the menu item whose number or reference is passed in *menu* and whose item number is passed in *menuItem*. You can pass -1 in *menuItem* in order to indicate the last item added to *menu*.

If you omit the *process* parameter, **Get menu item mark** applies to the menu bar for the current process. Otherwise, **Get menu item mark** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

If the menu item has no mark or if the *menuItem* parameter specifies a hierarchical submenu, **Get menu item mark** returns an empty string.

**Note:** See discussion of check marks on Macintosh and Windows in the description of the **SET MENU ITEM MARK** command.

## Example

The following example toggles the check mark of a menu item:

```
SET MENU ITEM MARK($vlMenu;$vlItem;Char(18)*Num(Character code(Get menu item mark($vlMenu;$vlItem))#18))
```

# Get menu item method

Get menu item method ( menu ; menuItem {; process} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu reference or Menu number |
| menuItem | Longint | ⇒ | Number of menu item or -1 for the last item added to the menu |
| process | Longint | ⇒ | Process number |
| Function result | String | ⇐ | Method name |

## Description

The **Get menu item method** command returns the name of the 4D project method associated with the menu item designated by the *menu* and *menuItem* parameters.

You can pass -1 in *menuItem* in order to specify the last item added to *menu*.

In *menu*, you can pass a menu reference (*MenuRef*) or a menu number. If you pass a menu reference, the *process* parameter is unnecessary and will be ignored if it is passed. If you pass a menu number, the command will take the corresponding menu in the main menu bar of the current process into account. If you want to designate another process, pass its number in the optional *process* parameter.

The command returns the name of the 4D method as a character string (expression). If no method is associated with a menu item, the command returns an empty string.

# Get menu item modifiers

Get menu item modifiers ( menu ; menuItem {; process} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu reference or Menu number |
| menuItem | Longint | ⇒ | Number of menu item or -1 for the last item added to the menu |
| process | Longint | ⇒ | Process number |
| Function result | Longint | ⇐ | Modification key(s) associated with menu item |

## Description

The **Get menu item modifiers** command returns any additional modifier(s) associated with the standard shortcut of the menu item designated by the *menu* and *menuItem* parameters.

The standard shortcut is composed of the Command (Mac OS) or Ctrl (Windows) key plus a custom key. The standard shortcut is managed using the **SET MENU ITEM SHORTCUT** and **Get menu item key** commands.

The additional modifiers are the Shift key and the Option (Mac OS) /Alt (Windows) key. These modifiers can only be used when a standard shortcut has been specified beforehand.

The number value returned by the command corresponds to the codeof the additional modifier key(s). The key codes are as follows:

- **Shift**= 512
- **Option** (Mac OS) or **Alt** (Windows) = 2048
  If both keys are used, their values are combined.

**Note:** You can evaluate the value returned using the Shift key mask and Option key mask constants of the "**Events (Modifiers)**" theme.

If the menu item does not have an associated modifier key, the command returns 0.

You can pass -1 in *menuItem* in order to specify the last item added to *menu*.

In *menu*, you can pass a menu reference (*MenuRef*) or a menu number.

If you pass a menu reference, the *process* parameter serves no purpose and will be ignored if it is passed.

If you pass a menu number, the command will take the corresponding menu in the main menu bar of the current process into account. If you want to designate another process, pass its number in the optional *process* parameter.

## Example

Refer to the example of the **Get menu item key** command.

# Get menu item parameter

Get menu item parameter ( menu ; menuItem ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu reference or Menu number |
| menuItem | Longint | ⇒ | Number of menu item or -1 for the last item added to the menu |
| Function result | String | ⇐ | Custom parameter of the menu item |

## Description

The **Get menu item parameter** command returns the custom character string associated with the menu item designated by the *menu* and *menuItem* parameters. This string must have been set beforehand using the **SET MENU ITEM PARAMETER** command.

# GET MENU ITEM PROPERTY

GET MENU ITEM PROPERTY ( menu ; menuItem ; property ; value {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu reference or Menu number |
| menuItem | Longint | ⇒ | Number of menu item or -1 for the last item added to the menu |
| property | String | ⇒ | Property type |
| value | Expression | ⇐ | Property value |
| process | Longint | ⇒ | Process number |

## Description

The **GET MENU ITEM PROPERTY** command returns, in the *value* parameter, the current value of the property of the menu item designated by the *menu* and *menuItem* parameters.

You can pass -1 in *menuItem* in order to specify the last item added to *menu*.

In *menu*, you can pass a menu reference (*MenuRef*) or a menu number. If you pass a menu reference, the *process* parameter is unnecessary and will be ignored if it is passed. If you pass a menu number, the command will take the corresponding menu in the main menu bar of the current process into account. If you want to designate another process, pass its number in the optional *process* parameter.

In the *property* parameter, pass the property for which you want to get the value. You can use one of the constants of the "**Menu Item Properties**" theme or a string corresponding to a custom property. For more information about menu properties and their values, please refer to the description of the **SET MENU ITEM PROPERTY** command.

# Get menu item style

Get menu item style ( menu ; menuItem {; process} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ➡ | Menu number or Menu reference |
| menuItem | Longint | ➡ | Menu item number or -1 for last item added |
| process | Longint | ➡ | Process reference number |
| Function result | Longint | ⮌ | Current menu item style |

## Description

The **Get menu item style** command returns the font style of the menu item whose number or reference is passed in *menu* and whose item number is passed in *menuItem*. You can pass -1 in *menuItem* in order to indicate the last item added to *menu*.

If you omit the *process* parameter, **Get menu item style** applies to the menu bar for the current process. Otherwise, **Get menu item style** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

**Get menu item style** returns a combination (one or a sum) of the following predefined constants, found in the **Font Styles** theme:

| Constant | Type | Value |
|---|---|---|
| Bold | Longint | 1 |
| Italic | Longint | 2 |
| Plain | Longint | 0 |
| Underline | Longint | 4 |

## Example

To test if a menu item is displayed in bold, you write:

```
If((Get menu item style($vlMenu;$vlItem)&Bold)#0)
   `...
End if
```

# GET MENU ITEMS

GET MENU ITEMS ( menu ; menuTitlesArray ; menuRefsArray )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇨ | Menu reference or Menu number |
| menuTitlesArray | String array | ⇦ | Array of menu titles |
| menuRefsArray | String array | ⇦ | Array of menu references |

## Description

The **GET MENU ITEMS** command returns, in the *menuTitlesArray* and *menuRefsArray* arrays, the titles and IDs of all the items of the menu or menu bar designated by the *menu* parameter.

In the *menu* parameter, you can pass a menu reference (*MenuRef*), a menu bar number or a menu bar reference obtained using the **Get menu bar reference** command.

If no menu reference is associated with an item, an empty string is returned in the corresponding array element.

## Example

You want to find out the contents of the menu bar of the current process:

```
ARRAY TEXT(menuTitlesArray;0)
ARRAY TEXT(menuRefsArray;0)
MenuBarRef:=Get menu bar reference(Frontmost process)
GET MENU ITEMS(MenuBarRef;menuTitlesArray;menuRefsArray)
```

# Get menu title

Get menu title ( menu {; process} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| process | Longint | ⇒ | Process reference number |
| Function result | String | ⇐ | Title of the menu |

## Description

The **Get menu title** command returns the title of the menu whose number or reference is passed in *menu.*

If you omit the *process* parameter, **Get menu title** applies to the menu bar for the current process. Otherwise, the command applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

# Get selected menu item parameter

Get selected menu item parameter -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String |  | Custom parameter of the menu item |

## Description

The **Get selected menu item parameter** command returns the custom character string assoaciated with the selected menu item. This parameter must have been set beforehand using the **SET MENU ITEM PARAMETER** command.

If no menu item has been selected, the command returns an empty string "".

# INSERT MENU ITEM

INSERT MENU ITEM ( menu ; afterItem ; itemText {; subMenu {; process}}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| afterItem | Longint | ⇒ | Menu item number |
| itemText | String | ⇒ | Text for the menu item to be inserted |
| subMenu | MenuRef | ⇒ | Reference of submenu associated with item |
| process | Longint | ⇒ | Process reference number |
| * | Operator | ⇒ | If passed: consider metacharacters as standard characters |

## Description

The **INSERT MENU ITEM** command inserts new menu items into the menu whose number or reference is passed in *menu* after the existing menu item whose number is passed in *afterItem*.

If you omit the *process* parameter, **INSERT MENU ITEM** applies to the menu bar for the current process. Otherwise, the command applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

If you do not pass the * parameter, **INSERT MENU ITEM** allows to you insert one or several menu items in one call.

**INSERT MENU ITEM** works like **APPEND MENU ITEM**, except that it enables you to insert items anywhere in the menu, while **APPEND MENU ITEM** always adds them at the end of the menu.

See the description of the **APPEND MENU ITEM** command for details about the item definition passed in *itemText* and about the action of the * parameter.

The optional *subMenu* parameter can be used to indicate a menu as the added item and thus position a hierarchical submenu. You must pass a menu reference (MenuRef type string) specifying a menu created, for example, using the **Create menu** command. If the command adds more than one menu item, the submenu is associated with the first item.

**Important:** The new items do not have any associated methods or actions. These must be associated with the items using the **SET MENU ITEM PROPERTY** or **SET MENU ITEM METHOD** commands, or the items can also be managed from within a form method using the **Menu selected** command.

## Example

The following example creates a menu consisting of two commands to which it assigns a method:

```
menuRef:=Create menu
APPEND MENU ITEM(menuRef;"Characters")
SET MENU ITEM METHOD(menuRef;1;"CharMgmtDial")
INSERT MENU ITEM(menuRef;1;"Paragraphs")
SET MENU ITEM METHOD(menuRef;2;"ParaMgmtDial")
```

# Menu selected

Menu selected {( subMenu )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| subMenu | MenuRef | ⇦ | Reference of menu containing item selected |
| Function result | Longint | ⇨ | Menu command selected Menu number in high word Menu item number in low word |

## Description

**Menu selected** is used only when forms are displayed. It detects which menu command has been chosen from a menu and, in the case of a hierarchical submenu, returns the reference of the submenu.

**Tip:** Whenever possible, use methods associated with menu commands in an associated menu bar (with a negative menu bar number) instead of using **Menu selected**. Associated menu bars are easier to manage, since it is not necessary to test for their selection.

The **Menu selected** command can be used to work with hierarchical submenus. When selecting a hierarchical menu item beyond the first level, the command returns, in the optional *subMenu* parameter, the reference (MenuRef type, 16-character string) of the submenu to which the selected item belongs. If the menu command does not contain a hierarchical submenu, this parameter receives an empy string.

**Menu selected** returns the menu-selected number, a long integer. To find the menu number, divide **Menu selected** by 65,536 and convert the result to an integer. To find the menu command number, calculate the modulo of **Menu selected** with the modulus 65,536. Use the following formulas to calculate the menu number and menu command number:

```
Menu:=Menu selected¥ 65536
menu command:=Menu selected% 65536
```

You can also extract these values using the *bitwise operators* as follows:

```
Menu:=(Menu selected & 0xFFFF0000)>>16
menu command:=Menu selected & 0xFFFF
```

If no menu commands are selected, **Menu selected** returns 0.

## Example

The following form method uses **Menu selected** to supply the menu and menu item arguments to **SET MENU ITEM MARK**:

```
Case of
   :(Form event=On Menu Selected)
      C_STRING(16;$refMenuIncludingItem)
      C_LONGINT($ref;$MenuNum;$MenuItemNum)
      $ref:=Menu selected($refMenuIncludingItem)
      $MenuNum:=$ref¥65536
      $MenuItemNum:=$ref%65536
      SET MENU ITEM MARK($refMenuIncludingItem;$MenuItemNum;Char(18))
End case
```

**Note:** The On Menu Selected form event is not activated if no item is selected, *$refMenuIncludingItem* is always given a value and *$MenuNum* equals 0 if the menu is not one of the menus of the menu bar.

# RELEASE MENU

RELEASE MENU ( menu )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | MenuRef | ➡ | Menu reference |

## Description

The **RELEASE MENU** command removes the menu whose ID is passed in *menu* from memory. This menu must have been created by the menu **Create menu** command. The following rule applies: for each **Create menu** there must be a corresponding **RELEASE MENU**.

The command removes every instance of the *menu* menu from every menu bar and every process. If the menu belongs to a menu bar which is in use, it will continue to work but can no longer be modified. It will only be truly removed from the memory when the last menu bar where it appears is no longer in use.

This command can be used with menus that are used as menu bars.

Any sub-menus used by *menu* are not removed if they were created directly using the **Create menu** command. In this case, you must remove them individually (see the rule mentioned above). However, if the submenus come from the duplication of an existing menu, do not call **RELEASE MENU** with them because 4D will erase them automatically.

## Example

This example shows different ways to use this command:

```
newMenu:=Create menu
APPEND MENU ITEM(newMenu;"Test1")
subMenu:=Create menu
APPEND MENU ITEM(subMenu;"SubTest1")
APPEND MENU ITEM(subMenu;"SubTest2") // Creation of items in submenu

APPEND MENU ITEM(newMenu;"Test2";subMenu) // Attaching submenu to menu

 //At this point, the submenu is attached to the menu and if we will not need it later on, this is right place to remove it.
 //Removing it does not immediately delete subMenu since it is still used by newMenu. subMenu is only deleted when newMenu
is.
 //Removing the submenu here lets you save memory
RELEASE MENU(subMenu)

$result1:=Dynamic pop up menu(newMenu) //Use of menu
copyMenu:=Create menu(newMenu) // Creation of menu by copying newMenu (and thus copying subMenu as well)
RELEASE MENU(newMenu) // We no longer need newMenu.

$result2:=Dynamic pop up menu(copyMenu)
RELEASE MENU(copyMenu)
 //You don't need to worry about deleting the submenus of copyMenu since it was not created directly using Create menu
 //The rule to follow is: each Create menu must have a corresponding RELEASE MENU
```

# SET MENU BAR

SET MENU BAR ( menuBar {; process}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| menuBar | Longint, String, MenuRef | ⇒ | Number or name of the menu bar or Menu reference |
| process | Longint | ⇒ | Process reference number |
| * | Operator | ⇒ | Save menu bar state |

## Description

**SET MENU BAR** replaces the current menu bar with the one specified by *menuBar* for the current process only. In the *menuBar* parameter, you can pass either the number or name of the new menu bar. You can also pass a menu ID (*MenuRef* type, 16-character string). When you work with references, the menus can be used as menu bars and vice versa (see the **Managing Menus** section).

**Note:** The name of a menu bar may contain up to 31 characters and must be unique.

The optional *process* parameter changes the menu bar of the specified process to *menuBar*.

**Note:** If you pass a *MenuRef* in *menuBar*, the *process* parameter serves no purpose and will be ignored.

The optional * parameter allows you to save the state of the menu bar. If this parameter is omitted, **SET MENU BAR** reinitializes the menu bar when the command is executed.

For example, suppose that **SET MENU BAR**(1) is executed. Next, several menu commands are disabled using the **DISABLE MENU ITEM** command.

If **SET MENU BAR(1)** is executed a second time, either from the same process or from a different process, all menu commands will revert to their initial enabled state.

If **SET MENU BAR(1;*)** is executed, the menu bar will retain the same state as before, and the menu commands that were disabled will remain disabled.

**Note:** If you pass a *MenuRef* in *menuBar*, the * parameter serves no purpose and will be ignored.

When a user enters the Application environment, the first menu bar is displayed (Menu Bar #1). You can change this menu bar when opening a database by specifying the desired menu bar in the **On Startup database method** or in the startup method for an individual user.

## Example 1

The following example changes the current menu bar to menu bar #3 and resets the states of the menu commands to their original states:

```
SET MENU BAR(3)
```

## Example 2

The following example changes the current menu bar to the menu bar named “FormMenuBar1” and saves the states of the menu commands. Menu commands that were previously disabled will appear disabled.

```
SET MENU BAR("FormMenuBar1";*)
```

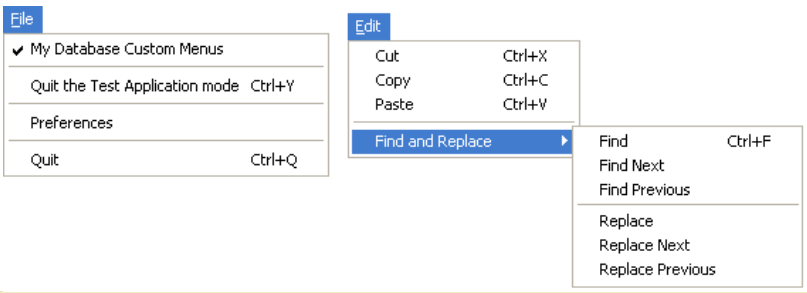
## Example 3

The following example sets the current menu bar to menu bar #3 while records are being modified. After the records have been modified, the menu bar is reset to menu bar #2, with the menu state saved:

```
SET MENU BAR(3)
ALL RECORDS([Customers])
MODIFY SELECTION([Customers])
SET MENU BAR(2;*)
```

## Example 4

In this comprehensive example, we will create, by programming, a menu bar including the following File and Edit menus:




```
 `Method for creating File menu
C_TEXT(FileMenu) ` FileMenu will contain the File menu reference
FileMenu:=Create menu
INSERT MENU ITEM(FileMenu;-1;"My Database "+Get indexed string(131;29))
SET MENU ITEM MARK(FileMenu;1;Char(18))
INSERT MENU ITEM(FileMenu;-1;"(-")
INSERT MENU ITEM(FileMenu;-1;"Quit the Test Application mode/Y")
SET MENU ITEM PROPERTY(FileMenu;3;Associated standard action;Return to Design mode)
INSERT MENU ITEM(FileMenu;-1;"(-")
INSERT MENU ITEM(FileMenu;-1;Get indexed string(131;26))
SET MENU ITEM PROPERTY(FileMenu;6;Associated standard action;Database settings action) `Settings
INSERT MENU ITEM(FileMenu;-1;"(-")
INSERT MENU ITEM(FileMenu;-1;Get indexed string(131;30))
SET MENU ITEM PROPERTY(FileMenu;7;Associated standard action;Quit action) `Quit
SET MENU ITEM SHORTCUT(FileMenu;7;Character code("Q"))

 `Method for creating Find and Replace menu
C_TEXT(FindAndReplaceMenu) `FindAndReplaceMenu will contain the Find and Replace menu reference
FindAndReplaceMenu:=Create menu
APPEND MENU ITEM(FindAndReplaceMenu;"Find;Find Next;Find Previous;(-;Replace;Replace Next;Replace Previous")
SET MENU ITEM SHORTCUT(FindAndReplaceMenu;1;Character code("F"))
SET MENU ITEM SHORTCUT(FindAndReplaceMenu;5;Character code("R"))
SET MENU ITEM METHOD(FindAndReplaceMenu;1;"MyFindMethod")

 `Method for creating Edit menu
C_TEXT(EditMenu) `EditMenu will contain the Edit menu reference
EditMenu:=Create menu
APPEND MENU ITEM(EditMenu;"Cut;Copy;Paste")
SET MENU ITEM SHORTCUT(EditMenu;1;Character code("X"))
SET MENU ITEM PROPERTY(EditMenu;1;Associated standard action;Cut action)
SET MENU ITEM SHORTCUT(EditMenu;2;Character code("C"))
SET MENU ITEM PROPERTY(EditMenu;2;Associated standard action;Copy action)
SET MENU ITEM SHORTCUT(EditMenu;3;Character code("V"))
SET MENU ITEM PROPERTY(EditMenu;3;Associated standard action;Paste action)
INSERT MENU ITEM(EditMenu;-1;"(-")
INSERT MENU ITEM(EditMenu;-1;"Find and Replace";FindAndReplaceMenu) ` item that will have submenu

main_Bar:=Create menu ` Create the menu bar made up of other menus
INSERT MENU ITEM(main_Bar;-1;Get indexed string(79;1);FileMenu)
APPEND MENU ITEM(main_Bar;"Edit";EditMenu)

SET MENU BAR(main_Bar)
```

# SET MENU ITEM

SET MENU ITEM ( menu ; menuItem ; itemText {; process}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| menuItem | Longint | ⇒ | Menu item number or -1 for the last item added |
| itemText | String | ⇒ | New text for the menu item |
| process | Longint | ⇒ | Process reference number |
| * | Operator | ⇒ | If passed: consider metacharacters as standard characters |

## Description

The **SET MENU ITEM** command changes the text of the menu item whose menu number or reference is passed in *menu* and whose item number is passed in *menuItem*, to the text passed in *itemText*. You can pass -1 in *menuItem* in order to designate the last item added to the *menu*.

If you do not pass the * parameter, any "special" characters included in *itemText* (such as ( ; or !) will be considered as instruction characters (metacharacters). For example, to set a menu item as a separator line, you insert the “-” character into *itemText*. If you pass the * parameter, the "special" characters will be considered as standard characters. Please refer to the description of the **APPEND MENU ITEM** command for more details about these characters.

If you omit the *process* parameter, **SET MENU ITEM** applies to the menu bar for the current process. Otherwise, **SET MENU ITEM** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

# SET MENU ITEM ICON

SET MENU ITEM ICON ( menu ; menuItem ; iconRef {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu reference or Menu number |
| menuItem | Longint | ⇒ | Number of menu item or -1 for the last item added to the menu |
| iconRef | Text, Longint | ⇒ | Name or number of picture to be associated with menu item |
| process | Longint | ⇒ | Process number |

## Description

The **SET MENU ITEM ICON** command modifies the icon associated with the menu item designated by the *menu* and *menuItem* parameters.
You can pass -1 in *menuItem* in order to specify the last item added to *menu*.

In *menu*, you can pass a menu reference (*MenuRef*) or a menu number. If you pass a menu reference, the command will apply to all the instances of the menu in all the processes. In this case, the *process* parameter is ignored if it is passed. If you pass a menu number, the command will take the corresponding menu in the main menu bar of the current process into account. If you want to designate another process, pass its number in the optional *process* parameter.

In *iconRef*, you can pass the picture to be used as the icon. You can use a library picture or a picture reference.

- Library picture: You can pass either the name or number of the picture. It is generally preferable to use its number rather than its name since picture numbers are unique IDs, which is not the case with names.
- Picture reference: The picture must be located in the **Resources** folder of the database and you must use a "file:{pathname}fileName" type syntax in *iconRef*. For more information about the **Resources** folder, refer to the **Resources** section.

## Example

Use of a picture located in the Resources folder of the database:

```
SET MENU ITEM ICON($MenuRef;2;"File:English.lproj/spot.png")
```

# SET MENU ITEM MARK

SET MENU ITEM MARK ( menu ; menuItem ; mark {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| menuItem | Longint | ⇒ | Item number or -1 for last item added |
| mark | String | ⇒ | New menu item mark |
| process | Longint | ⇒ | Process reference number |

## Description

The **SET MENU ITEM MARK** command changes the check mark of the menu item whose menu number or reference is passed in *menu* and whose item number is passed in *menuItem* to the first character of the string passed in *mark*. You can pass -1 in *menuItem* in order to designate the last item added to the *menu*.

If you omit the *process* parameter, **SET MENU ITEM MARK** applies to the menu bar for the current process. Otherwise, the command applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

If you pass an empty string, any mark is removed from the menu item. Otherwise:

- On Macintosh, the first character of the string becomes the mark of the menu item. Usually, you will pass **Char(18)**, which is the check mark character for Macintosh menus.
- On Windows, the menu item is assigned the standard check mark.

## Example

See example for the **Get menu item mark** command.

# SET MENU ITEM METHOD

SET MENU ITEM METHOD ( menu ; menuItem ; methodName {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu reference or Menu number |
| menuItem | Longint | ⇒ | Number of menu item or -1 for the last item added to the menu |
| methodName | String | ⇒ | Method name |
| process | Longint | ⇒ | Process number |

## Description

The **SET MENU ITEM METHOD** command modifies the 4D project method associated with the menu item designated by the *menu* and *menuItem* parameters.

You can pass -1 in *menuItem* in order to specify the last item added to *menu*.

In *menu*, you can pass a menu reference (*MenuRef*) or a menu number. If you pass a menu reference, the command will apply to all the instances of the menu in all the processes. In this case, the *process* parameter is ignored if it is passed. If you pass a menu number, the command will be applied to the corresponding menu in the main menu bar of the current process. If you want to designate another process, pass its number in the optional *process* parameter.

In *method*, pass the name of the 4D method as a character string (expression).

**Note:** If the menu item corresponds to the title of a hierarchical sub-menu, the method will not be called when the menu item is selected.

## Example

Refer to the example of the **SET MENU BAR** command.

# SET MENU ITEM PARAMETER

SET MENU ITEM PARAMETER ( menu ; menuItem ; param )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu reference or Menu number |
| menuItem | Longint | ⇒ | Number of menu item or -1 for the last item added to the menu |
| param | String | ⇒ | String to associate as parameter |

## Description

The **SET MENU ITEM PARAMETER** command associates a custom character string with a menu item designated by the *menu* and *menuItem* parameters.

This parameter is mainly used by the **Dynamic pop up menu** command.

## Example

This code provides a menu including the names of the open windows and lets you get the number of the window chosen:

```
WINDOW LIST($alWindow)
$tMenuRef:=Create menu
For($i;1;Size of array($alWindow))
   APPEND MENU ITEM($tMenuRef;Get window title($alWindow{$i})) // Title of menu item
   SET MENU ITEM PARAMETER($tMenuRef;-1;String($alWindow{$i})) // Value returned by menu item
End for
$tWindowRef:=Dynamic pop up menu($tMenuRef)
RELEASE MENU($tMenuRef)
```

# SET MENU ITEM PROPERTY

SET MENU ITEM PROPERTY ( menu ; menuItem ; property ; value {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu reference or menu number |
| menuItem | Longint | ⇒ | Number of menu item or -1 for the last item added to the menu |
| property | String | ⇒ | Property type |
| value | Expression | ⇒ | Property value |
| process | Longint | ⇒ | Process number |

## Description

The **SET MENU ITEM PROPERTY** command sets the *value* of the *property* for the menu item designated by the *menu* and *menuItem* parameters.

You can pass -1 in *menuItem* to specify the last item added to *menu*.

In *menu*, you can pass a menu reference (*MenuRef*) or a menu number. If you pass a menu reference, the command will apply to all the instances of the menu in all the processes. In this case, the *process* parameter is ignored if it is passed. If you pass a menu number, the command will take the corresponding menu in the main menu bar of the current process into account. If you want to designate another process, pass its number in the optional *process* parameter.

In the *property* parameter, pass the property whose value you want to modify and pass the new value in *value*. For the *property* parameter, you can use one of the constants of the "**Menu Item Properties**" theme or any custom value:
**Standard property**: The constants of the "**Menu Item Properties**" theme as well as their possible values are described below. Note that in the case of the Associated standard action property, you can pass one of the constants of the "**Value for Associated Standard Action**" theme in the *value* parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Access privileges | String | 4D_access_group | Assign an access group to the command<br>0 = All Groups<br>>0 = Group ID |
| Associated standard action | String | 4D_standard_action | Associate a standard action with a menu item<br>See the constants of the **Value for Associated Standard Action** theme |
| Start a new process | String | 4D_start_new_process | Activate the "Start New Process" option<br>0 = No, 1 = Yes |

Here are the constants for the "**Value for Associated Standard Action**" theme:

| Constant | Type | Value |
|---|---|---|
| Accept action | Longint | 2 |
| Add subrecord action | Longint | 14 |
| Cancel action | Longint | 1 |
| Clear action | Longint | 21 |
| Copy action | Longint | 19 |
| Cut action | Longint | 18 |
| Database settings action | Longint | 32 |
| Delete record action | Longint | 7 |
| Delete subrecord action | Longint | 13 |
| Edit subrecord action | Longint | 12 |
| First page action | Longint | 10 |
| First record action | Longint | 5 |
| Last page action | Longint | 11 |
| Last record action | Longint | 6 |
| MSC action | Longint | 36 |
| Next page action | Longint | 8 |
| Next record action | Longint | 3 |
| No action | Longint | 0 |
| Paste action | Longint | 20 |
| Previous page action | Longint | 9 |
| Previous record action | Longint | 4 |
| Quit action | Longint | 27 |
| Redo action | Longint | 31 |
| Return to Design mode | Longint | 35 |
| Select all action | Longint | 22 |
| Show clipboard action | Longint | 23 |
| Test application action | Longint | 26 |
| Undo action | Longint | 17 |

For more information about standard menu item properties, refer to the "**Building menus**" section of the Design Reference manual.

**Custom property**: In *property*, you can pass any custom text and associate a *value* of the text, number or Boolean type with it. This *value* will be stored with the item and can be retrieved using the **GET MENU ITEM PROPERTY** command. You can use any custom string in the *property* parameter, simply make sure not to use a title used by 4D (by convention, properties set by 4D begin with "4D_").

**Note:** If the menu item corresponds to the title of a hierarchical sub-menu, the standard action will not be called when the menu item is selected.

# SET MENU ITEM SHORTCUT

SET MENU ITEM SHORTCUT ( menu ; menuItem ; itemKey ; modifiers {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| menuItem | Longint | ⇒ | Menu item number or -1 for last item added |
| itemKey | String, Longint | ⇒ | Letter of keyboard shortcut or Character code of keyboard shortcut (former syntax) |
| modifiers | Longint | ⇒ | Modifier(s) to associate with shortcut (ignored if key code is passed) |
| process | Longint | ⇒ | Process reference number |

## Description

The **SET MENU ITEM SHORTCUT** command replaces the shortcut key associated with the menu command specified by *menu* and *menuItem*, by the character whose character code or text is passed in *itemKey*. You can pass -1 in *menuItem* in order to indicate the last item added to *menu*. This key is combined with the **Ctrl** (Windows) or **Command** (Macintosh) key in order to set the new keyboard shortcut.

You can pass the letter indicating the shortcut key in the *itemKey* parameter, for example “U” to specify the **Ctrl+U** (Windows) or **Command+U** (Mac OS) shortcut. You can pass additional modifiers to associate with the shortcut in the *modifiers* parameter. This way you can define shortcuts like **Ctrl+Alt+Shift+Z** (Windows) or **Cmd+Option+Shift+Z** (Mac OS).

You can pass the following values in *modifiers*:

- 256 for the **Command** (Mac OS) or **Ctrl** (Windows) key
- 512 for the **Shift** key
- 2048 for the **Option** (Mac OS) or **Alt** (Windows) key
- To associate several different keys, just combine their values.

**Note:** You can specify the value to pass using the Command key mask, Shift key mask and Option key mask constants of the **Events (Modifiers)** theme.

The **Ctrl** (Windows) and **Command** (Mac OS) keys are automatically added by 4D to the keyboard shortcut, regardless of whether or not you explicitly indicate it in the *modifiers* parameter. So you do not have to add the value 256 to this parameter, unless this key is the only modifier, in which case you must pass either the value 256 or the corresponding constant in *modifiers*.

**Note:** For compatibility, the command also accepts a character code in the *itemKey* parameter (former syntax). In this case, the *modifiers* parameter is not taken into account and can be omitted. The shortcut is only associated with the **Ctrl** (Windows) or **Command** (Mac OS) key.

If you omit the *process* parameter, **SET MENU ITEM SHORTCUT** applies to the menu bar for the current process. Otherwise, **SET MENU ITEM SHORTCUT** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

If you pass 0 (zero) in *itemKey*, any shortcut is removed from the menu item.

## Example 1

Definition of the Ctrl+Shift+U (Windows) and Cmd+Shift+U (Mac OS) shortcut for the “Underline” menu item:

```
SET MENU ITEM(menuRef;1;"Underline")
SET MENU ITEM SHORTCUT(menuRef;1;"U";Shift key mask)
```

## Example 2

Definition of the Ctrl+R (Windows) and Cmd+R (Mac OS) shortcut for the "Restart" menu item:

```
INSERT MENU ITEM(FileMenu;-1;"Restart")
SET MENU ITEM SHORTCUT(FileMenu;-1;"R";Command key mask)
```

# SET MENU ITEM STYLE

SET MENU ITEM STYLE ( menu ; menuItem ; itemStyle {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| menu | Longint, MenuRef | ⇒ | Menu number or Menu reference |
| menuItem | Longint | ⇒ | Menu item number or -1 for last item added |
| itemStyle | Longint | ⇒ | New menu item style |
| process | Longint | ⇒ | Process reference number |

## Description

The **SET MENU ITEM STYLE** command changes the font style of the menu item whose menu number or reference is passed in *menu* and whose item number is passed in *menuItem* according to the font style passed in *itemStyle*. You can pass -1 in *menuItem* in order to indicate the last item added to *menu*.

If you omit the *process* parameter, **SET MENU ITEM STYLE** applies to the menu bar for the current process. Otherwise, **SET MENU ITEM STYLE** applies to the menu bar for the process whose reference number is passed in *process*.

**Note:** If you pass a *MenuRef* in *menu*, the *process* parameter serves no purpose and will be ignored.

You specify the font style of the item in the *itemStyle* parameter. You pass a combination (one or a sum) of the following predefined constants, found in the **Font Styles** theme:

| Constant | Type | Value |
|---|---|---|
| Bold | Longint | 1 |
| Italic | Longint | 2 |
| Plain | Longint | 0 |
| Underline | Longint | 4 |

# Messages

- ALERT
- CONFIRM
- DISPLAY NOTIFICATION
- GOTO XY
- MESSAGE
- MESSAGES OFF
- MESSAGES ON
- Request

# ALERT

ALERT ( message ; okButtonTitle ; test )

| Parameter | Type | | Description |
|---|---|---|---|
| message | String | ⇒ | Message to display in the alert dialog box |
| okButtonTitle | String | ⇒ | OK button title |
| test | 2D Text array | ⇒ | entrée plat dessert |

## Description

The **ALERT** command displays an alert dialog box composed of a note icon, a message, and an OK button.

You pass the message to be displayed in the parameter *message*. This message can be up to 255 characters long. However, if the message does not fit into the message area, it can appear truncated, depending on its length and the width of the characters.

By default, the title of the OK button is “OK.” To change the title of the OK button, pass the new custom title into the optional parameter *okButtonTitle*. If necessary, the OK button width is resized toward the left, according to the width of the custom title you pass.

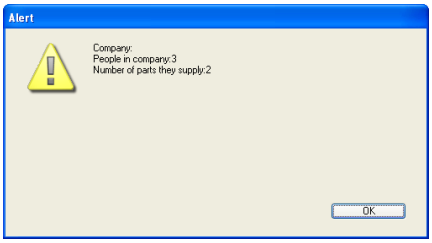
**Tip:** Do not call the **ALERT** command from the section of a form or object method that handles the On Activate or On Deactivate form events; this will cause an endless loop.

## Example 1

This example displays an alert showing information about a company. Note that the displayed string contains carriage returns, which cause the string to wrap to the next line:

```
ALERT("Company: "+[Companies]Name+Char(13)+"People in company: "+
String(Records in selection([People]))+Char(13)+"Number of parts they supply: "+
String(Records in selection([Parts])))
```

This line of code displays the following alert box (on Windows):

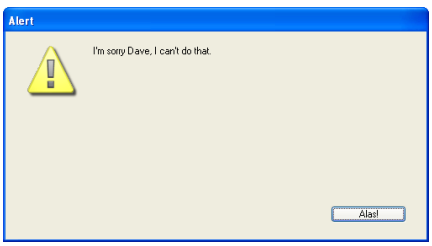



## Example 2

The line:

```
ALERT("I'm sorry Dave, I can't do that.";"Alas!")
```

displays the alert dialog box (on Windows) shown:

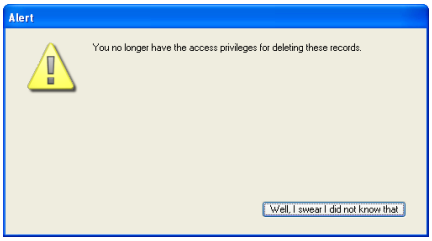



## Example 3

The line:

```
ALERT("You no longer have the access privileges for deleting these records.":"Well, I swear I did not know that")
```

displays the alert dialog box (on Windows) shown:



# CONFIRM

CONFIRM ( message {; okButtonTitle {; cancelButtonTitle}} )

| Parameter | Type | | Description |
|---|---|---|---|
| message | String | ⇒ | Message to display in the confirmation dialog box |
| okButtonTitle | String | ⇒ | OK button title |
| cancelButtonTitle | String | ⇒ | Cancel button title |

## Description

The **CONFIRM** command displays a confirm dialog box composed of a note icon, a message, an OK button, and a Cancel Button.

You pass the message to be displayed in the *message* parameter. This message can be up to 255 characters long. If the message does not fit into the message area, it can appear truncated, depending on its length and the width of the characters.

By default, the title of the OK button is “OK” and that of the Cancel button is “Cancel.” To change the titles of these buttons, pass the new custom titles into the optional parameters *okButtonTitle* and *cancelButtonTitle*. If necessary, the width of the buttons is resized toward the left, according to the width of the custom titles you pass.

The OK button is the default button. If the user clicks the OK button or presses Enter to accept the dialog box, the OK system variable is set to 1. If the user clicks the Cancel button to cancel the dialog box, the OK system variable is set to 0.

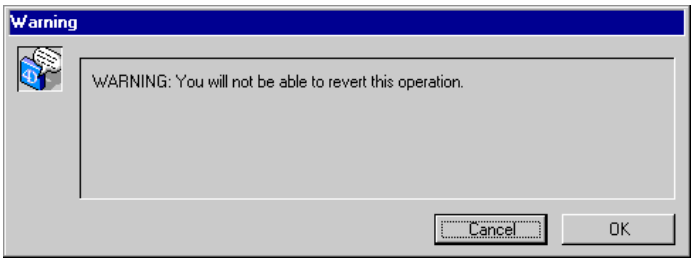
**Tip:** Do not call the **CONFIRM** command from the section of a form or object method that handles the On Activate or On Deactivate form events; this will cause an endless loop.

## Example 1

The line:

```
CONFIRM("WARNING: You will not be able to revert this operation.")
If(OK=1)
   ALL RECORDS([Old Stuff])
   DELETE SELECTION([Old Stuff])
Else
   ALERT("Operation canceled.")
End if
```

will display the confirm dialog box (on Windows) shown here:

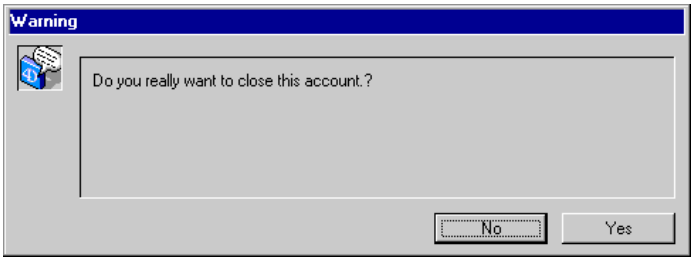



## Example 2

The line:

```
CONFIRM("Do you really want to close this account?";"Yes";"No")
```

will display the confirm dialog box (on Windows) shown here:

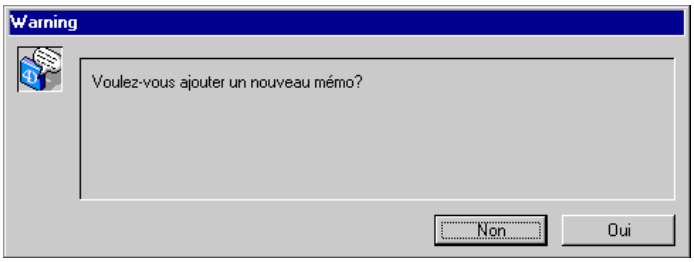



## Example 3

You are writing a 4D application for the international market. You wrote a project method that returns the correct localized text from its English version. You have also populated an array named *<>asLocalizedUIMessages,* where you store the most common words. In doing so, the line:

```
CONFIRM(INTL Text("Do you want to add a new Memo?");<>asLocalizedUIMessages{kLoc_YES};
<>asLocalizedUIMessages{kLoc_NO})
```

could display the French confirm dialog box (on Windows) shown here:

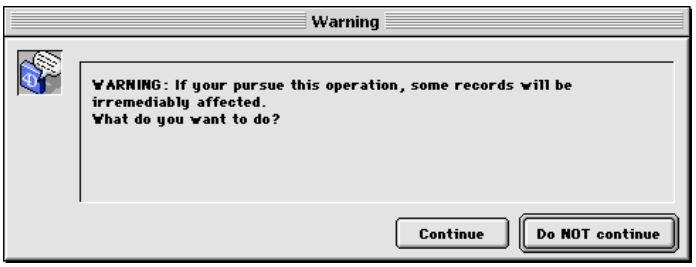



## Example 4

The line:

```
CONFIRM("WARNING: If your pursue this operation, some records will be "+"irremediably affected."+Char(13)+
"What do you want to do?";"Do NOT continue";"Continue")
```

will display the confirm dialog box (on Macintosh) shown here:



# DISPLAY NOTIFICATION

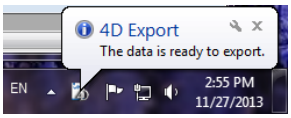
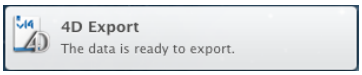
DISPLAY NOTIFICATION ( title ; text {; duration} )

| Parameter | Type | | Description |
|---|---|---|---|
| title | Alpha | ➡ | Notification title |
| text | Alpha | ➡ | Notification text |
| duration | Longint | ➡ | Display duration in seconds |

## Description

The **DISPLAY NOTIFICATION** command displays a notification message to the user.

Usually this kind of message is used by the OS or an application to inform the user of an external event (network disconnection, availability of an upgrade, etc.).

- Under Windows, the message appears in the notification area of the taskbar:
  
  
- Under OS X (version 10.8 minimum), the message appears in a small sliding window in the top right corner of the screen.
  
  
  Note that in compliance with Apple specifications, the notification is only displayed when the application is not in the foreground. However, the message still appears in the "notification center" list.

In *title* and *text*, pass the title and the text of the message to display (in our example, the title is “4D Export”). You can enter up to 255 characters.

Under Windows, the message window remains displayed as long as no activity has been detected on the machine, or until the user clicks on the close box. The optional *duration* parameter modifies the default display duration. Note that the display of notifications depends on the system configuration.

## Example

```
DISPLAY NOTIFICATION("4D Export";"The data is ready to export.")
```

# GOTO XY

GOTO XY ( x ; y )

| Parameter | Type | | Description |
|---|---|---|---|
| x | Longint | ⇒ | x (horizontal) position of cursor |
| y | Longint | ⇒ | y (vertical) position of cursor |

## Description

The **GOTO XY** command is used in conjunction with the **MESSAGE** command when you display messages in a window opened using **Open window**.

**GOTO XY** positions the character cursor (an invisible cursor) to set the location of the next message in the window.

The upper-left corner is position 0,0. The cursor is automatically placed at 0,0 when a window is opened and after **ERASE WINDOW** is executed.

After **GOTO XY** positions the cursor, you can use **MESSAGE** to display characters in the window.

## Example 1

See example for the **MESSAGE** command.

## Example 2

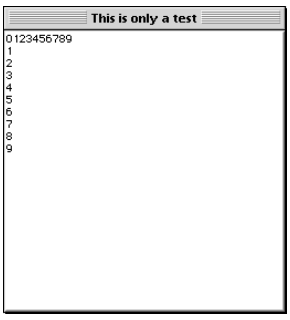
See example for the **Milliseconds** command.

## Example 3

The following example:

```
Open window(50;50;300;300;5;"This is only a test")
For($vlRow;0;9)
   GOTO XY($vlRow;0)
   MESSAGE(String($vlRow))
End for
For($vlLine;0;9)
   GOTO XY(0;$vlLine)
   MESSAGE(String($vlLine))
End for
$vhStartTime:=Current time
Repeat
Until((Current time-$vhStartTime)>?00:00:30?)
```

displays the following window (on Macintosh) for 30 seconds:



# MESSAGE

MESSAGE ( message )

| Parameter | Type | | Description |
|---|---|---|---|
| message | String | ➡ | Message to display |

## Description

The **MESSAGE** command is usually used to inform the user of some activity. It displays *message* on the screen in a special message window that opens and closes each time you call **MESSAGE**, unless you work with a window you previously opened using **Open window** (see the following details). The message is temporary and is erased as soon as a form is displayed or the method stops executing. If another **MESSAGE** is executed, the old message is erased.

If a window is opened with **Open window**, all subsequent calls to **MESSAGE** display the messages in that window. The window behaves like a terminal:

- Successive messages do not erase previous messages when displayed in the window. Instead, they are concatenated onto existing messages.
- If a message is wider than the window, 4D automatically performs text wrap.
- If a message has more lines than the window, 4D automatically scrolls the message window.
- To control line breaks, include carriage returns — **Char(13)** — into your message.
- To display the text at a particular place in the window, call **GOTO XY**.
- To erase the contents of the window, call **ERASE WINDOW**.
- The window is only an output window and does not redraw when other windows overlap it.
- You can modify the font and size of characters displayed in the window by means of the "Interface" page in the Database Settings.

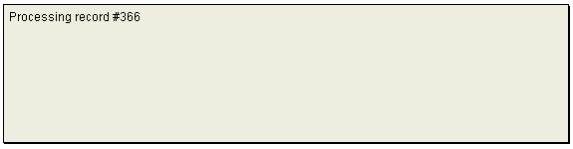
**Note: MESSAGE** is compatible with the **Open form window** command; however, in this context the second * parameter of **Open form window**, which saves the window's size and position, is not supported.

## Example 1

The following example processes a selection of records and calls **MESSAGE** to inform the user about the progress of the operation:

```
For($vlRecord;1;Records in selection([anyTable]))
   MESSAGE("Processing record #"+String($vlRecord))
   ` Do Something with the record
   NEXT RECORD([anyTable])
End for
```

The following window appears and disappears at each **MESSAGE** call:

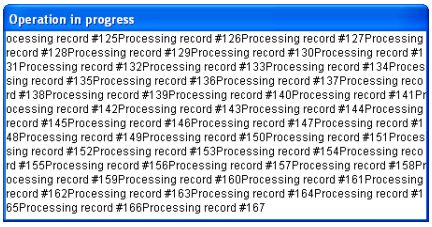



## Example 2

In order to avoid this "blinking" window, you can display the messages in a window opened using **Open window**, as in this example:

```
Open window(50;50;500;250;5;"Operation in Progress")
For($vlRecord;1;Records in selection([anyTable]))
   MESSAGE("Processing record #"+String($vlRecord))
   ` Do Something with the record
   NEXT RECORD([anyTable])
End for
CLOSE WINDOW
```

This provides the following result (shown here on Windows):

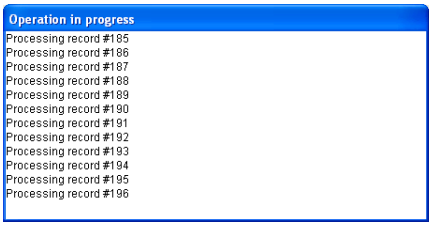



## Example 3

Adding a carriage return makes a better presentation:

```
Open window(50;50;500;250;5;"Operation in Progress")
For($vlRecord;1;Records in selection([anyTable]))
   MESSAGE("Processing record #"+String($vlRecord)+Char(Carriage return))
   ` Do Something with the record
   NEXT RECORD([anyTable])
End for
CLOSE WINDOW
```

This provides the following result (shown here on Windows):

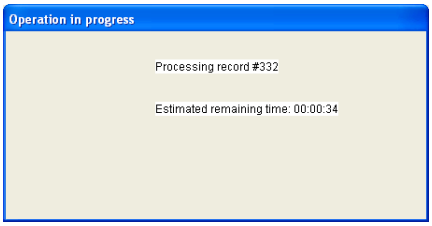



## Example 4

Using **GOTO XY** and writing some additional lines:

```
Open window(50;50;500;250;5;"Operation in Progress")
$vlNbRecords:=Records in selection([anyTable])
$vhStartTime:=Current time
For($vlRecord;1;$vlNbRecords)
   GOTO XY(5;2)
   MESSAGE("Processing record #"+String($vlRecord)+Char(Carriage return))
   ` Do Something with the record
   NEXT RECORD([anyTable])
   GOTO XY(5;5)
   $vlRemaining:=(($vlNbRecords/$vlRecord)-1)*(Current time-$vhStartTime)
   MESSAGE("Estimated remaining time: "+Time string($vlRemaining))
End for
CLOSE WINDOW
```

This provides the following result (shown here on Windows):



# MESSAGES OFF

MESSAGES OFF
Does not require any parameters

## Description

The **MESSAGES OFF** and **MESSAGES ON** commands turn on and off the progress meters displayed by 4D while executing time-consuming operations. By default, messages are on.

The following table shows operations that display the progress meter:

| | | |
|---|---|---|
| Apply Formula | Quick Report | Order by |
| Export Data | Import Data | Graph |
| Query by Form | Query by Formula | Query Editor |

The following table lists the commands that display the progress meter:

**APPLY TO SELECTION**
**Average**
**BUILD APPLICATION**
**DISTINCT VALUES**
**EXPORT DIF**
**EXPORT SYLK**
**EXPORT TEXT**
**_o_GRAPH TABLE**
**IMPORT DIF**
**IMPORT SYLK**
**IMPORT TEXT**
**Max**
**Min**
**ORDER BY**
**ORDER BY FORMULA**
**QR REPORT**
**QUERY**
**QUERY BY FORMULA**
**QUERY BY EXAMPLE**
**QUERY SELECTION**
**QUERY SELECTION BY FORMULA**
**REDUCE SELECTION**
**RELATE MANY SELECTION**
**RELATE ONE SELECTION**
**SCAN INDEX**
**Sum**

**Note for 4D Server:** Starting with 4D Server v14 R3, progress message windows are no longer shown on the server since these operations are automatically listed on the **Real Time Monitor Page** of the administration window. If you want to force these progress windows to be displayed, you must call the **MESSAGES ON** command on the server.

## Example

The following example turns off the progress meter before doing a sort, and then turns it back on after completing the sort:

```
MESSAGES OFF
ORDER BY([Addresses];[Addresses]ZIP;>;[Addresses]Name2;>)
MESSAGES ON
```

# MESSAGES ON

MESSAGES ON

Does not require any parameters

## Description

See the description of the **MESSAGES OFF** command.

# Request

Request ( message {; defaultResponse {; OKButtonTitle {; CancelButtonTitle}}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| message | String | ➡ | Message to display in the request dialog box |
| defaultResponse | String | ➡ | Default data for the enterable text area |
| OKButtonTitle | String | ➡ | OK button title |
| CancelButtonTitle | String | ➡ | Cancel button title |
| Function result | String | ↩ | Value entered by user |

## Description

The **Request** command displays a request dialog box composed of a message, a text input area, an **OK** button, and a **Cancel** Button.

You pass the message to be displayed in the *message* parameter. If the message does not fit in the display area (usually around 50 characters, but it varies depending on the System and the font used), it can appear truncated.

By default, the title of the **OK** button is “OK” and that of the **Cancel** button is “Cancel.” To change the titles of these buttons, pass the new custom titles into the optional parameters *OKButtonTitle* and *CancelButtonTitle*. If necessary, the width of the buttons is resized toward the left, according to the width of the custom titles you pass.

The **OK** button is the default button. If you click the **OK** button or press **Enter** to accept the dialog box, the OK system variable is set to 1. If you click the **Cancel** button to cancel the dialog box, the OK system variable is set to 0.

The user can enter text into the text input area. To specify a default value, pass the default text in the *defaultResponse* parameter. If the user clicks **OK**, **Request** returns the text. If the user clicks **Cancel**, **Request** returns an empty string (""). If the response should be a numeric or a date value, convert the string returned by **Request** to the proper type with the **Num** or **Date** functions.

**Tip:** Do not call the **Request** command from the section of a form or object method that handles the On Activate or On Deactivate form event; this will cause an endless loop.

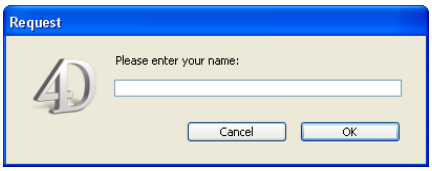
**Tip:** If you need to get several pieces of information from the user, design a form and present it with **DIALOG**, rather than presenting a succession of **Request** dialog boxes.

## Example 1

The line:

```
$vsPrompt:=Request("Please enter your name:")
```

will display the request dialog box (on Windows) shown here:

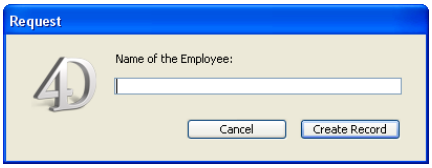



## Example 2

The line:

```
vsPrompt:=Request("Name of the Employee:";"";"Create Record";"Cancel")
If(OK=1)
   ADD RECORD([Employees])
   ` Note: vsPrompt is then copied into the field [Employees]Last name
   ` during the On Load event in the form method
End if
```

will display the request dialog box (on Windows) shown here:

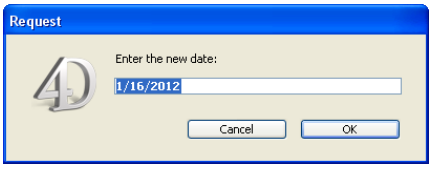



## Example 3

The line:

```
$vdPrompt:=Date(Request("Enter the new date:";String(Current date)))
```

will display the request dialog box (on Windows) shown here:



# Named Selections

- Named Selections
- CLEAR NAMED SELECTION
- COPY NAMED SELECTION
- CUT NAMED SELECTION
- USE NAMED SELECTION

# Named Selections

Named selections provide an easy way to manipulate several selections simultaneously. A named selection is an ordered list of records for a table in a process. This ordered list can be given a name and kept in memory. Named selections offer a simple means to preserve in memory the order of the selection and the current record of the selection.

The following commands enable you to work with named selections:

- **COPY NAMED SELECTION**
- **CUT NAMED SELECTION**
- **USE NAMED SELECTION**
- **CLEAR NAMED SELECTION**
- **CREATE SELECTION FROM ARRAY**

Named selections are created with the **COPY NAMED SELECTION**, **CUT NAMED SELECTION** and **CREATE SELECTION FROM ARRAY** commands. Named selections are generally used to work on one or more selections and to save and later restore an ordered selection. There can be many named selections for each table in a process. To reuse a named selection as the current selection, call **USE NAMED SELECTION**. When you are done with a named selection, use **CLEAR NAMED SELECTION**.

**Note:** Combining the statement **SET QUERY DESTINATION**(Into named selection;namedselection) with a search command (for example **QUERY**) can also be used to create a named selection. Refer to the description of the **SET QUERY DESTINATION** command.

Named selections can be local, process or interprocess in scope.

A named selection is local when its name is preceded by a dollar sign. When its name is not preceded by any symbol, it is a process named selection and it is an interprocess named selection if its name is preceded by the symbols (<>) — a “less than” sign followed by a “greater than” sign.

**Note**: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (**Option-Shift-V** on US keyboard).

The scope of an interprocess named selection is identical to the scope of an interprocess variable. An interprocess named selection can be accessed from any process.
With 4D in remote mode and 4D Server, an interprocess named selection is available only to the processes of the client that created it. An interprocess named selection is not available to other client machines.

A process named selection is available only within the process in which it was created and on the server.

A local named selection is defined for the process that created it and is not visible on the server.

**Warning**: Creating a named selection requires access to the selection of the table. Since selections are kept on the server and a local process does not have access to server data, do not use named selections within local processes.

## Visibility of Sets

The following table indicates the principles concerning the visibility of named selections depending on their scope and where they were created:

| | Client Process | Other client processes | Other clients | Server process | Other server processes |
|---|---|---|---|---|---|
| **Creation in a client process** | | | | | |
| $test | x | | | | |
| test | x | | | x (Trigger) | |
| <>test | x | x | | | |
| **Creation in a server process** | | | | | |
| $test | | | | x | |
| test | | | | x | |
| <>test | | | | x | x |

## Named Selections and Sets

The differences between sets and named selections are:

- A named selection is an ordered list of records; a set is not.
- Sets are very memory efficient, because they require only one bit for each record in the file. Named selections require 4 bytes for each record in the selection.

- Unlike sets, named selections cannot be saved to disk.
- Sets have the standard **INTERSECTION**, **UNION** and **DIFFERENCE** operations; named selections cannot be combined with other named selections.

The similarities between named selections and sets are:

- Like a set, a named selection exists in memory.
- A named selection and a set store references to a record. If records are modified or deleted, the named selection or the set can become invalid.
- Like a set, a named selection “remembers” the current record as of the time the named selection was created.

# CLEAR NAMED SELECTION

CLEAR NAMED SELECTION ( name )

| Parameter | Type | | Description |
|---|---|---|---|
| name | String | ⇨ | Name of named selection to be cleared |

## Description

**CLEAR NAMED SELECTION** clears *name* from memory and frees the memory used by *name*. The command does not affect tables, selections, or records. Since named selections use memory, it is good practice to clear named selections when they are no longer needed.

If *name* was created using the **CUT NAMED SELECTION** command and then manipulated using the **USE NAMED SELECTION** command, *name* no longer exists in memory. In this case, the **CLEAR NAMED SELECTION** command does not need to be used.

# COPY NAMED SELECTION

COPY NAMED SELECTION ( {aTable ;} name )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to copy selection, or Default table, if omitted |
| name | String | ⇒ | Name of the named selection to create |

## Description

**COPY NAMED SELECTION** copies the current selection of *aTable* to the named selection *name*. The default table for the process is used if the optional *table* parameter is not specified. The parameter *name* contains a copy of the selection. The current selection and the current record of *aTable* for the process are not changed.

A named selection does not actually contain the records, but only an ordered list of references to records. Each reference to a record takes 4 bytes in memory. This means that when a selection is copied using the **COPY NAMED SELECTION** command, the amount of memory required is 4 bytes multiplied by the number of records in the selection. Since named selections reside in memory, you should have enough memory for the named selection as well as the current selection of the table in the process.

Use the **CLEAR NAMED SELECTION** command to free the memory used by *name*.

## Example

The following example allows you to check if there are other overdue invoices in the *[People]* table. The selection is sorted and then saved. We search for all records where invoices are due. Then we reuse the selection and clear the named selection in memory. Clearing the named selection in memory is optional, in case the database designer wants to keep the sorted selection for future use:

```
ALL RECORDS([People])
 `Allow the user to sort the selection
ORDER BY([People])
 ` Save the sorted selection as a named selection
COPY NAMED SELECTION([People];"UserSort")
 ` Search for records where invoices are due
QUERY([People];[People]InvoiceDue=True)
 ` If records are found
If(Records in selection([People])>0)
 ` Alert the user
   ALERT("Yes, there are overdue invoices on table.")
End if
 ` Reuse the sorted named selection
USE NAMED SELECTION("UserSort")
 ` Remove the selection from memory
CLEAR NAMED SELECTION("UserSort")
```

# CUT NAMED SELECTION

CUT NAMED SELECTION ( {aTable ;} name )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to cut selection, or Default table, if omitted |
| name | String | ⇒ | Name of the named selection to create |

## Description

**CUT NAMED SELECTION** creates a named selection *name* and moves the current selection of *aTable* to it. This command differs from **COPY NAMED SELECTION** in that it does not copy the current selection, but moves the current selection of *table* itself.

After the command has been executed, the current selection of *aTable* in the current process becomes empty. Therefore, **CUT NAMED SELECTION** should not be used while a record is being modified.

**CUT NAMED SELECTION** is more memory efficient than **COPY NAMED SELECTION**. With **COPY NAMED SELECTION**, 4 bytes times the number of selected records is duplicated in memory. With **CUT NAMED SELECTION**, only the reference to the list is moved.

## Example

The following method empties the current selection of a table *[Customers]*:

```
CUT NAMED SELECTION([Customers];"ToBeCleared")
CLEAR NAMED SELECTION("ToBeCleared")
```

# USE NAMED SELECTION

USE NAMED SELECTION ( name )

| Parameter | Type | | Description |
|---|---|---|---|
| name | String | ⇨ | Name of named selection to be used |

## Description

**USE NAMED SELECTION** uses the named selection *name* as the current selection for the table to which it belongs.

When you create a named selection, the current record is “remembered” by the named selection. **USE NAMED SELECTION** retrieves the position of this record and makes the record the new current record; this command loads the current record. If the current record was modified after *name* was created, the record should be saved before **USE NAMED SELECTION** is executed, in order to avoid losing the modified information.

- If **COPY NAMED SELECTION** was used to create *name*, the named selection *name* is copied to the current selection of the table to which *name* belongs. The named selection *name* exists in memory until it is cleared. Use the **CLEAR NAMED SELECTION** command to clear the named selection and free the memory used by *name*.
- If **CUT NAMED SELECTION** was used to create *name*, the current selection is set to *name* and *name* no longer exists in memory.

Remember that a named selection is a representation of a selection of records at the moment that the named selection is created. If the records represented by the named selection change, the named selection may no longer be accurate. Therefore, a named selection represents a group of records that does not change frequently. A number of things can invalidate a named selection: modifying a record of the named selection, deleting a record of the named selection, or changing the criterion that determined the named selection.

# Objects (Forms)

## 4D Write Pro areas

Most commands of the "**Objects (Forms)**" theme support 4D Write Pro areas. For more information about this, refer to the **Using commands from the Objects (Forms) theme** section in the 4D Write Pro Reference manual.

- Object Properties
- GET STYLE SHEET INFO
- LIST OF STYLE SHEETS
- OBJECT DUPLICATE
- OBJECT Get action
- OBJECT Get auto spellcheck
- OBJECT GET BEST SIZE
- OBJECT Get border style
- OBJECT Get context menu
- OBJECT GET COORDINATES
- OBJECT Get corner radius
- OBJECT Get data source
- OBJECT GET DRAG AND DROP OPTIONS
- OBJECT Get enabled
- OBJECT Get enterable
- OBJECT GET EVENTS
- OBJECT Get filter
- OBJECT Get focus rectangle invisible
- OBJECT Get font
- OBJECT Get font size
- OBJECT Get font style
- OBJECT Get format
- OBJECT Get help tip
- OBJECT Get horizontal alignment
- OBJECT Get indicator type
- OBJECT Get keyboard layout
- OBJECT Get list name
- OBJECT Get list reference
- OBJECT GET MAXIMUM VALUE
- OBJECT GET MINIMUM VALUE
- OBJECT Get multiline
- OBJECT Get name
- OBJECT Get placeholder
- OBJECT Get pointer
- OBJECT GET PRINT VARIABLE FRAME
- OBJECT GET RESIZING OPTIONS
- OBJECT GET RGB COLORS
- OBJECT GET SCROLL POSITION
- OBJECT GET SCROLLBAR
- OBJECT GET SHORTCUT
- OBJECT Get style sheet
- OBJECT GET SUBFORM
- OBJECT GET SUBFORM CONTAINER SIZE
- OBJECT Get text orientation
- OBJECT Get three states checkbox
- OBJECT Get title
- OBJECT Get type

- OBJECT Get vertical alignment
- OBJECT Get visible
- OBJECT Is styled text
- OBJECT MOVE
- OBJECT SET ACTION
- OBJECT SET AUTO SPELLCHECK
- OBJECT SET BORDER STYLE
- OBJECT SET COLOR
- OBJECT SET CONTEXT MENU
- OBJECT SET COORDINATES
- OBJECT SET CORNER RADIUS
- OBJECT SET DATA SOURCE
- OBJECT SET DRAG AND DROP OPTIONS
- OBJECT SET ENABLED
- OBJECT SET ENTERABLE
- OBJECT SET EVENTS
- OBJECT SET FILTER
- OBJECT SET FOCUS RECTANGLE INVISIBLE
- OBJECT SET FONT
- OBJECT SET FONT SIZE
- OBJECT SET FONT STYLE
- OBJECT SET FORMAT
- OBJECT SET HELP TIP
- OBJECT SET HORIZONTAL ALIGNMENT
- OBJECT SET INDICATOR TYPE
- OBJECT SET KEYBOARD LAYOUT
- OBJECT SET LIST BY NAME
- OBJECT SET LIST BY REFERENCE
- OBJECT SET MAXIMUM VALUE
- OBJECT SET MINIMUM VALUE
- OBJECT SET MULTILINE
- OBJECT SET PLACEHOLDER
- OBJECT SET PRINT VARIABLE FRAME
- OBJECT SET RESIZING OPTIONS
- OBJECT SET RGB COLORS
- OBJECT SET SCROLL POSITION
- OBJECT SET SCROLLBAR
- OBJECT SET SHORTCUT
- OBJECT SET STYLE SHEET
- OBJECT SET SUBFORM
- OBJECT SET TEXT ORIENTATION
- OBJECT SET THREE STATES CHECKBOX
- OBJECT SET TITLE
- OBJECT SET VERTICAL ALIGNMENT
- OBJECT SET VISIBLE
- *_o_DISABLE BUTTON*
- *_o_ENABLE BUTTON*

# Object Properties

The Object Properties commands act on the properties of objects present in forms. They enable you to change the appearance and behavior of the objects while using the forms to display records and in the Application environment.

**Important**: The scope of these commands is the form currently being used; changes disappear when you exit the form.

## Accessing Objects using their Object Names or their Data Source Names

The object property commands share the same generic syntax described here:

**COMMAND NAME**({*;} object { ; additional parameters specific to each command )

If you specify the optional * parameter, you indicate an object name (a string) in *object*.

You can use the @ character within that name if you want to address several objects of the form in one call. The following table shows examples of object names you can specify to this command.

| Object Names | Objects affected by the call |
|---|---|
| mainGroupBox | Only the object *mainGroupBox*. |
| main@ | The objects whose name starts with “main”. |
| @GroupBox | The objects whose name ends with “GroupBox”. |
| @Group@ | The objects whose name contains “Group”. |
| main@Btn | The objects whose name starts with “main” and ends with “Btn”. |
| @ | All the objects present in the form. |

Form object names can contain up 255 bytes, allowing you to define and apply custom naming rules, such as "xxxx_Button" or "xxx_Mac".

**Note**: You can configure the way the @ character is interpreted when it is included in a character string. This option affects the functioning of the commands in the "Objects (Forms)" theme. For more information, refer to the 4D *Design Reference* manual.

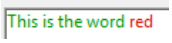
If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string.

## Interaction of generic commands with multi-style areas

Starting with 4D v14, a new way of interacting has been defined between generic commands such as **OBJECT SET RGB COLORS** or **OBJECT SET FONT STYLE** and multi-style text areas.

In previous versions of 4D, executing one of these commands modified the contents of any custom style tags inserted in the area. From now on, only default properties are affected by these commands (as well as any properties saved by means of default tags). Custom style tags remain as they are.

For example, given a multi-style area where default tags were saved:




The plain text of the area is as follows:

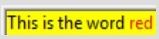
```
<span style="text-align:left;font-family:'Segoe UI';font-size:9pt;color:#009900">This is the word <span
style="color:#D81E05">red</span></span>
```

If you execute the following code:

```
OBJECT SET COLOR(*;"myArea";-(Blue+(256*Yellow)))
```

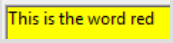
With 4D v14, the red color remains:

**4D v14 and higher**




```
<span style="text-align:left;font-
family:'Segoe UI';font-
size:9pt;color:#0000FF">This is the word
<span style="color:#D81E05">red</span>
</span>
```

**prior versions**




```
<span style="font-family:'Segoe UI';font-size:9pt;text-
align:left;font-weight:normal;font-style:normal;text-
decoration:none;color:#0000FF;"><span style="background-
color:#FFFFFF">This is the word red</span></span>
```

The following generic commands are concerned:

**OBJECT SET RGB COLORS**
**OBJECT SET COLOR**
**OBJECT SET FONT**
**OBJECT SET FONT STYLE**
**OBJECT SET FONT SIZE**

In the context of multi-style areas, generic commands should be used to set default styles only. To manage styles during database execution, we recommend using the commands of the "**Styled Text**" theme.

# GET STYLE SHEET INFO

GET STYLE SHEET INFO ( styleSheetName ; font ; size ; styles )

| Parameter | Type | | Description |
|---|---|---|---|
| styleSheetName | Text | ⇒ | Name of style sheet |
| font | Text | ⇐ | Character font |
| size | Longint | ⇐ | Font size |
| styles | Longint | ⇐ | Style value |

## Description

The **GET STYLE SHEET INFO** command returns the current configuration of the style sheet designated in the *styleSheetName* parameter.

In *styleSheetName*, you pass the name of the style sheet as defined in the Design mode. To designate an automatic style sheet, you can use one of the following constants, found in the "**Font Styles**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Automatic style sheet | String | __automatic__ | Used by default for all objects |
| Automatic style sheet_additional | String | __automatic_additional_text__ | Supported by static text, fields and variables only. Used for additional text in dialog boxes. |
| Automatic style sheet_main | String | __automatic_main_text__ | Supported by static text, fields and variables only. Used for main text in dialog boxes. |

In *font*, the command returns the name of the font associated with the style sheet for the current platform.

In *size*, the command returns the size in points of the font associated with the style sheet for the current platform.

In *styles*, the command returns a value corresponding to the style(s) associated with the style sheet for the current platform. You can compare the value received with the following constants, found in the "**Font Styles**" theme:

| Constant | Type | Value |
|---|---|---|
| Bold | Longint | 1 |
| Bold and Italic | Longint | 3 |
| Bold and Underline | Longint | 5 |
| Italic | Longint | 2 |
| Italic and Underline | Longint | 6 |
| Plain | Longint | 0 |
| Underline | Longint | 4 |

If the command is executed correctly, the *OK* system variable is set to 1. Otherwise (for example, if the *styleSheetName* does not exist), it is set to 0.

## Example

You want to find out the current configuration of the "Automatic" style sheet:

```
C_LONGINT($size;$style)
C_TEXT($font)
GET STYLE SHEET INFO(Automatic style sheet;$font;$size;$style)
```

# LIST OF STYLE SHEETS

LIST OF STYLE SHEETS ( arrStyleSheets )

| Parameter | Type | | Description |
|---|---|---|---|
| arrStyleSheets | Text array | ⇦ | Names of style sheets defined in the application |

## Description

The **LIST OF STYLE SHEETS** command returns the list of application style sheets in the *arrStyleSheets* array.

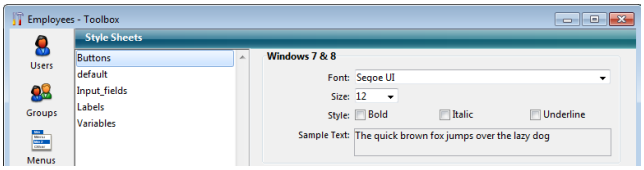
If it was not already defined previously, the *arrStyleSheets* text array is created by the command. It is automatically sized according to the number of style sheets defined.

After executing the command, each element of the array contains the name of a style sheet. These names are sorted alphabetically, as in the style sheet editor. The first array element always contains "__automatic__", which represents the "Automatic" style sheet.

**Note:** For compatibility reasons, the automatic style sheets "__automatic_main_text__" and "__automatic_additional_text__" are not returned by this command. However, they are still available in the forms.

## Example

In your application, the following style sheets are defined:




If you execute the following code:

```
LIST OF STYLE SHEETS($arrStyles)
 // $arrStyles{1} contains "__automatic__"
 // $arrStyles{2} contains "Buttons"
 // $arrStyles{3} contains "default"
 // $arrStyles{4} contains "Input_fields"
 // $arrStyles{5} contains "Labels"
 // $arrStyles{6} contains "Variables"
```

# OBJECT DUPLICATE

OBJECT DUPLICATE ( {* ;} object {; newName {; newVar {; boundTo {; moveH {; moveV {; resizeH {; resizeV}}}}}}} {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| newName | Text | ⇒ | Name of new object |
| newVar | Pointer | ⇒ | Pointer to variable of new object |
| boundTo | Text | ⇒ | Name of previous enterable object (or radio button) |
| moveH | Longint | ⇒ | Horizontal shift of new object (>0 = to the right, <0 = to the left) |
| moveV | Longint | ⇒ | Vertical shift of new object (>0 = downwards, <0 = upwards) |
| resizeH | Longint | ⇒ | Value of the horizontal resize of the object |
| resizeV | Longint | ⇒ | Value of the vertical resize of the object |
| * | Operator | ⇒ | If specified= absolute coordinates If omitted= relative coordinates |

## Description

The **OBJECT DUPLICATE** command is used to create a copy of the object designated by the *object* parameter in the context of the form being executed (Application mode). The source form, generated in Design mode, is not modified.

By default, all the options specified in the Property list for the source object are applied to the copy (size, resizing options, color, etc.), including any associated object method.
However, the following exceptions should be noted:

- Default button: there can only be one default button in a form. When you duplicate a button having the "Default button" property, this property is assigned to the copy and is removed from the source object.
- Keyboard equivalents: the keyboard shortcut associated with a source object is not duplicated. This property is left blank in the copy.
- Object names: there cannot be several objects with the same name in a form. If you do not pass the *newName* parameter, the name of the source object is automatically incremented in the new object (see below).

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.
If you pass a field or variable reference and if the form contains several objects that use the same reference, the first occurrence found will be used. In this case, in order to avoid any ambiguity, it is recommended to work with object names, that are unique.

Pass the name assigned to the copy of the object in the *newName* parameter. This name must be in keeping with the rules for naming objects and be unique in the form. If it is not valid or already used by another object, the command does nothing and the *OK* variable returns 0.
If you omit this parameter or pass an empty string, the new name is automatically generated by incrementing the source object name (if this name is not already used). For example:

| Source name | Name of copy |
|---|---|
| Button | Button1 |
| Button20 | Button21 |
| Button21 | Button23 if Button22 already exists |

Pass a pointer to the variable to be associated with the new object in *newVar*. As a rule, you must point to a variable of the same type as the that of the source object but certain kinds of "retyping" are possible. The command provides automatic functions to facilitate writing generic code:

- Usually, all "enterable" variables can be retyped; for example, an object displaying a Date or Longint can be duplicated and used with a variable of the Text type. Any compatible properties will be kept. The command also permits changing types between Text objects and Picture objects. Note that a text object that is duplicated and associated with a Boolean variable or field will automatically appear as a check box.
- It is usually possible to dynamically transform a variable into a field and vice versa.
  On the other hand, graphic objects (buttons, check boxes, and so on) cannot be transformed into other types of controls.

If the variable type is not compatible with the object, the command does nothing and the *OK* variable is set to 0. If you omit this parameter, the variable is created dynamically by 4D (see the paragraph **Dynamic variables**). If you duplicate a static object (lines, rectangle, static picture, etc.), this parameter is ignored. Pass a Nil (->[]) pointer if you want to be able to use the other parameters.

You use the *boundTo* parameter in two cases:

- update of entry order: in this case, in *boundTo*, pass the name of the enterable object located just before the duplicated object. If you want for the new object to become the first object in the entry order of the page, pass the new Object First in entry order constant (see the **OBJECT Get pointer** command).
- association with a group of radio buttons: radio buttons function in a coordinated fashion when they are grouped. If the duplicated object is a radio button, in *boundTo* pass the name of a radio button of the group to which you want to attach the new object.

If you omit this parameter or pass an empty string, the new object becomes the last enterable object of the form page. In the case of a radio button, the object is attached to the group of the source button.

The new object can be moved and resized via the *moveH*, *moveV*, *resizeH* and *resizeV* parameters. As with the **OBJECT MOVE** command, the direction of the move or the resizing is specified by the sign of the values passed in the *moveH* and *moveV* parameters:

- If the value is positive, the move or resizing is carried out, respectively, to the right or downwards.
- If the value is negative, the move or resizing is carried out, respectively, to the left or upwards.

By default, the values of *moveH*, *moveV*, *resizeH* and *resizeV* modify the coordinates of the object in relation to its previous position. If you want for these parameters to specify absolute coordinates, pass the optional final * parameter.
If you omit these parameters, the new object is superimposed on top of the source object.

This command must be used in the context of the display of a form. It will generally be called in the On Load form event or following a user action (On Clicked event).

**Note:** If the On Load form event is associated with the source object, it is generated for the duplicated object when the command is executed. This allows, for example, the value of the object to be initialized.

For technical and logical reasons, **OBJECT DUPLICATE** cannot be called within the certain form events, in particular:

- On Load event generated in an object method
- On Unload event
- Event related to printing context (On Header, On Printing Detail, etc.). To print an object several times, you must use the **Print object** command.

When the command is called in a context that is not supported,the object is not duplicated and the *OK* variable is set to 0. If it is called in a printing context, the error -10601 is generated as well.

If the command was executed correctly, the *OK* variable is set to 1. Otherwise, it is set to 0.

## Example 1

Creation of a new button named "CancelButton" on top of the existing "OKButton" object and association with the *vCancel* variable:

```
OBJECT DUPLICATE(*;"OKButton";"CancelButton";vCancel)
```

## Example 2

Creation of a new radio button "bRadio6" based on the existing radio button "bRadio5". This button will be associated with the variable <>r6, integrated with the group of the "bRadio5" button and placed 20 pixels above it:

```
OBJECT DUPLICATE(*;"bRadio5";"bRadio6";<>r6;"bRadio5";0;20)
```

# OBJECT Get action

OBJECT Get action ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Text | ⇐ | Associated standard action |

## Description

The **OBJECT Get action** command returns the code of the standard action associated with the object(s) designated by the *object* and * parameters.

You can set a standard action for an object in the Design mode using the Property List (see **Standard actions** in the *Design Reference* manual), or using the **OBJECT SET ACTION** command.

The command returns a string containing the code for the standard action associated with the object. You can compare the value received with the constants of the "**Text Values for Associated Standard Action**" theme:

| Constant | Type | Value |
|---|---|---|
| Object Accept action | String | 2 |
| Object Add subrecord action | String | 14 |
| Object Automatic splitter action | String | 16 |
| Object Cancel action | String | 1 |
| Object Clear action | String | 21 |
| Object Copy action | String | 19 |
| Object Cut action | String | 18 |
| Object Database Settings action | String | 32 |
| Object Delete record action | String | 7 |
| Object Delete subrecord action | String | 13 |
| Object Edit subrecord action | String | 12 |
| Object First page action | String | 10 |
| Object First record action | String | 5 |
| Object Goto page action | String | 15 |
| Object Last page action | String | 11 |
| Object Last record action | String | 6 |
| Object MSC action | String | 36 |
| Object Next page action | String | 8 |
| Object Next record action | String | 3 |
| Object No standard action | String | 0 |
| Object Open back URL action | String | 37 |
| Object Open next URL action | String | 38 |
| Object Paste action | String | 20 |
| Object Previous page action | String | 9 |
| Object Previous record action | String | 4 |
| Object Quit action | String | 27 |
| Object Redo action | String | 31 |
| Object Refresh current URL action | String | 39 |
| Object Return to Design mode action | String | 35 |
| Object Select all action | String | 22 |
| Object Show Clipboard action | String | 23 |
| Object Stop loading URL action | String | 40 |
| Object Test Application action | String | 26 |
| Object Undo action | String | 17 |

## Example

You want to associate the "Cancel" action with all the objects in the form that do not already have any associated action:

```
ARRAY TEXT($arrObjects;0)

FORM GET OBJECTS($arrObjects)
For($i;1;Size of array($arrObjects))
   If(OBJECT Get action(*;$arrObjects{$i})=Object No standard action)
      OBJECT SET ACTION(*;$arrObjects{$i};Object Cancel action)
   End if
End for
```

# OBJECT Get auto spellcheck

OBJECT Get auto spellcheck ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable or field |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable or field (if * is omitted) |
| Function result | Boolean | ⇐ | True = automatic spell-checking,<br>False = no automatic spell-checking |

## Description

The **OBJECT Get auto spellcheck** command returns the status of the Auto spellcheck option for the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable or a field. In this case, you pass a reference instead of a name.

The command returns **True** when automatic spell-checking is enabled for the *object* and **False** when it is not.

# OBJECT GET BEST SIZE

OBJECT GET BEST SIZE ( {* ;} object ; bestWidth ; bestHeight {; maxWidth} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified = object is an object name (String) If omitted = object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or Field or variable (if * is omitted) |
| bestWidth | Longint | ⬅ | Optimum object width |
| bestHeight | Longint | ⬅ | Optimum object height |
| maxWidth | Longint | ➡ | Maximum object width |

## Description

The **OBJECT GET BEST SIZE** command returns the *bestWidth* and *bestHeight* parameters, the “optimal” width and height of the form object designated by the * and *object* parameters. These values are expressed in pixels. This command is particularly useful for displaying or printing complex reports, associated with the **OBJECT MOVE** command.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a character string). If you do not pass the * parameter, this indicates that *object* is a field or a variable. In this case, do not pass a string but rather a field or variable reference (object type only).

The optimal values returned indicate the minimum size of the object so that its current contents are entirely included within the limits. Usually these values are only meaningful for objects containing text. This calculation takes the font, font size, font style and object contents into account. It also takes hyphenation and carriage returns into consideration. Note that in the case of 3D buttons, the command works even when button contains only an icon. If the object specified is empty, the *bestWidth* returned is 0.

The size returned does not take into account any graphic frame applied around the object, nor any scrollbars. To obtain the real size of an object on screen, it is necessary to add the width of these elements.

The optional *maxWidth* parameter enables you to attribute a maximum width to the object. If the optimal width of the object is greater than this value, **OBJECT GET BEST SIZE** returns *maxWidth* in the *bestWidth* parameter and increases the optimal height as a consequence.

The following objects are handled by this command:

- Static text areas
- Text inserted in the form of references
- Fields and variables of the following types: Alpha, Text, Real, Integer, Long Integer, Date, Time, Boolean (check boxes and radio buttons)
- Buttons
- List box columns in display context (only visible rows are taken into account).

For all other form object types (group areas, tabs, rectangles, straight lines, circles/ovals, external areas, etc.), the **OBJECT GET BEST SIZE** command returns the current object size (defined in the form editor and possibly using the **OBJECT MOVE** command).

## Example

Refer to the example in the **SET PRINT MARKER** command.

# OBJECT Get border style

OBJECT Get border style ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Longint | ⇐ | Border line style |

## Description

The **OBJECT Get border style** command returns the border line style of the object(s) designated by the *object* and * parameters.

You can set the border line style for an object in Design mode using the Property List, or using the **OBJECT SET BORDER STYLE** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

The command returns a value corresponding to the border line style. You can compare the value received with the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Border Dotted | Longint | 2 | Objects appear framed with a dotted 1-pt. border line |
| Border Double | Longint | 5 | Objects appear framed with a double line, i.e., two continuous 1-pt. lines separated by a pixel |
| Border None | Longint | 0 | Objects appear with no border |
| Border Plain | Longint | 1 | Objects appear framed with a continuous 1-pt. border line |
| Border Raised | Longint | 3 | Objects appear framed with a 3D effect (raised) |
| Border Sunken | Longint | 4 | Objects appear framed with a sunken 3D effect |
| Border System | Longint | 6 | The border line is drawn based on the graphic specifications of the system |

# OBJECT Get context menu

OBJECT Get context menu ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Boolean | ⇐ | True = context menu enabled, False = context menu disabled |

## Description

The **OBJECT Get context menu** command returns the current state of the "Context Menu" option for the object(s) designated by the *object* and * parameters.

You can set the "Context Menu" option in Design mode using the Property List, or using the **OBJECT SET CONTEXT MENU** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, this indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

The command returns **True** if the context menu is enabled for the object and **False** otherwise.

# OBJECT GET COORDINATES

OBJECT GET COORDINATES ( {* ;} object ; left ; top ; right ; bottom )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified = object is the name of the object (string) If omitted = object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| left | Longint | ⇐ | Left coordinate of the object |
| top | Longint | ⇐ | Top coordinate of the object |
| right | Longint | ⇐ | Right coordinate of the object |
| bottom | Longint | ⇐ | Bottom coordinate of the object |

## Description

The **OBJECT GET COORDINATES** command returns the coordinates *left*, *top*, *right* and *bottom* (in points) in variables or fields of the object(s) of the current form defined by the parameters * and *object*.

If you pass the optional parameter *, it indicates that the object parameter is an object name (a string). If you don't pass the optional parameter *, it indicates that object is a field or a variable. In this case, you don't pass a string but a field or variable reference (only a field or variable of type object).

If you pass an object name to *object* and use the wildcard character ("@") to select more than one object, the coordinates returned will be those of the rectangle formed by all the objects concerned.

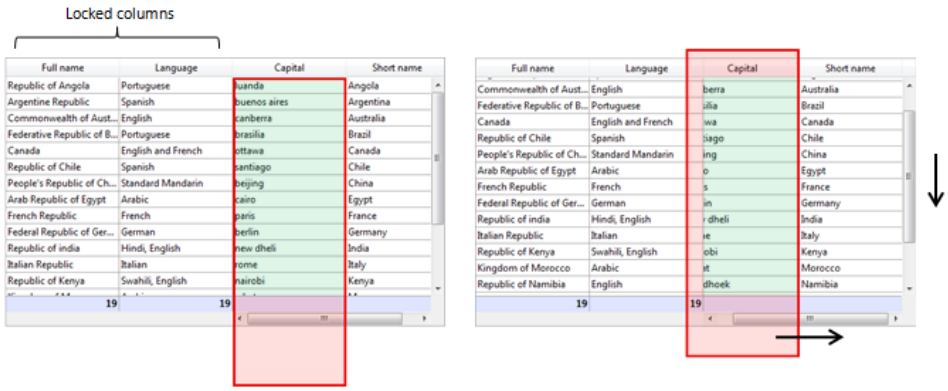
**Note:** Since 4D version 6.5, it is possible to set the interpretation mode of the wildcard character ("@"), when it is included in a string of characters. This option has an impact on the "Object Properties" commands. Please refer to the 4D Design Reference manual.

If the object doesn't exist or if the command is not called in a form, the coordinates (0;0;0;0) are returned.

In the context of list boxes, the **OBJECT GET COORDINATES** command can return the coordinates of specific list box parts, i.e. columns, headers, or footers, and not just those of the list box parent object. In 4D versions prior to v14 R5, this command always returned the parent list box coordinates, regardless of the area passed as parameter. From now on, when the referenced *object* is a list box header, column, or footer sub-object, the coordinates returned are those of the designated list box sub-object. You can use this new feature, for example, to display a small icon in the list box header cell when it is hovered over, which a user can then click to display a context menu.
For consistency, the reference frame used is the same when the object is a list box sub-object or a list box object: the origin is the upper-left corner of the form which contains the object. For list box sub-objects, the coordinates returned are theoretical; they take into account the scrolling state of the list box before any clipping occurs (i.e., the cutting carried out according to the coordinates of the parent list box). As a result, the sub-object may not be visible (or only partially so) at its coordinates, and these coordinates may be outside the form limits (or even negative). To find out whether the sub-object is visible (and which part of it is visible) you need to compare the coordinates returned with the list box coordinates, while considering the following rules:

- All sub-objects are clipped to the coordinates of their parent list box (as returned by **OBJECT GET COORDINATES** on the list box).
- Header and footer sub-objects are displayed on top of column content: when the coordinates of a column intersect the coordinates of the header or footer rows, then the column is not displayed at this intersection.
- Elements of locked columns are displayed on top of elements of scrollable columns: when the coordinates of an element in a scrollable column intersect the coordinates of an element in a locked column, then it is not displayed at this intersection.

For example, consider the following graphic where the coordinates of the *Capital* column are symbolized by the red rectangle:




As you can see in the first picture, the column is larger than the list box, so its coordinates go beyond the lower limit of the list box, including the footer. In the second picture, the list box has scrolled, so the column has also been moved "under" the *Language* column and header area. In any case, in order to calculate the actual visible part (green area), you need to subtract the red areas.

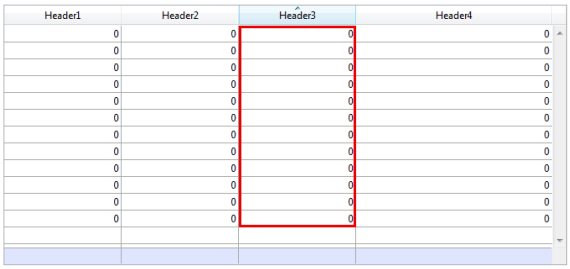
## Example 1

Let's assume that you want to obtain the coordinates of a rectangle formed by all the objects that begin with "button":

```
OBJECT GET COORDINATES(*;"button@";vLeft;vTop;vRight;vBottom)
```

## Example 2

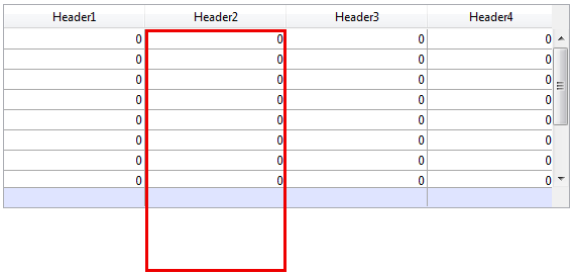
For interface needs, you want to surround the clicked area with a red rectangle:




In the object method of the list box, you can write:

```
OBJECT SET VISIBLE(*;"rectangleInfo";False) //initialize a red rectangle
$ptr:=OBJECT Get pointer(Object current)
OBJECT GET COORDINATES($ptr->;$x1;$y1;$x2;$y2)
OBJECT SET VISIBLE(*;"RedRect";True)
OBJECT SET COORDINATES(*;"RedRect";$x1;$y1;$x2;$y2)

OBJECT GET COORDINATES(*;"LB1";$lbx1;$lby1;$lbx2;$lby2)
If($lby1>$y1)|($lby2<$y2) // if the clicked area is outside the list box
   OBJECT SET VISIBLE(*;"Alert";True) //display a warning
Else
   OBJECT SET VISIBLE(*;"Alert";False)
End if
```

The method returns theoretical coordinates. In cases where the list box has been resized, you may need to calculate the clipping to know which part is visible:



# OBJECT Get corner radius

OBJECT Get corner radius ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| Function result | Longint | ⇐ | Radius of rounded corners (in pixels) |

## Description

The **OBJECT Get corner radius** command returns the current value of the corner radius for the rounded rectangle object whose name is passed in the *object* parameter. This value may have been set at the form level using the Property list (see **Corner radius (rectangles)**), or for the current process using the **OBJECT SET CORNER RADIUS** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

**Note:** In the current version of 4D, this command only applies to rounded rectangles (which are static objects). As a result, only the syntax based on the object name (using the * parameter) is supported.

This command returns the radius of rounded corners in pixels. By default, this value is 5 pixels.

## Example

The following code could be added to a button method:

```
C_LONGINT($radius)
$radius:=OBJECT Get corner radius(*;"GreenRect") //get current value
OBJECT SET CORNER RADIUS(*;"GreenRect";$radius+1) //increase radius
 // Maximum value will be handled automatically:
 // when reached, the button no longer has any effect
```

# OBJECT Get data source

OBJECT Get data source ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Pointer | ⇐ | Pointer to current data source of object |

## Description

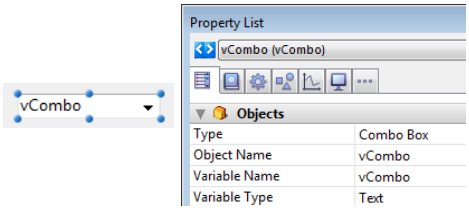
The **OBJECT Get data source** command returns the current data source of the object(s) designated by the *object* and * parameters.

You can define the data source for an object in Design mode using the Property List, or using the **OBJECT SET DATA SOURCE** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

## Example

Given a combo box object defined in a form:




You execute the following code:

```
$vPtr :=OBJECT Get data source(*;"vCombo")
 // $vPtr contains -> vCombo
```

# OBJECT GET DRAG AND DROP OPTIONS

OBJECT GET DRAG AND DROP OPTIONS ( {* ;} object ; draggable ; automaticDrag ; droppable ; automaticDrop )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| draggable | Boolean | ⇐ | Draggable = True; otherwise, False |
| automaticDrag | Boolean | ⇐ | Automatic Drag = True; otherwise, False |
| droppable | Boolean | ⇐ | Droppable = True; otherwise, False |
| automaticDrop | Boolean | ⇐ | Automatic Drop = True; otherwise, False |

## Description

The **OBJECT GET DRAG AND DROP OPTIONS** command returns the drag and drop options for the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

The command returns the current drag and drop options, as set in Design mode or for the current process using the **OBJECT SET DRAG AND DROP OPTIONS** command.

Each parameter returns True or False according to whether the corresponding option is enabled or disabled:

- *draggable* = True: Object draggable in programmed mode.
- *automaticDrag* = True (only used with text fields and variables, combo boxes and list boxes): Object draggable in automatic mode.
- *droppable* = True: Object accepts drops in programmed mode.
- *automaticDrop* = True (only used with picture fields and variables, text, combo boxes and list boxes): Object accepts drops in automatic mode.

# OBJECT Get enabled

OBJECT Get enabled ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | Boolean | ⇐ | True = object(s) enabled; Otherwise, false |

## Description

The **OBJECT Get enabled** command returns True if the object or group of objects designated by *object* is enabled in the form and False if it is not enabled.

An enabled object reacts to mouse clicks and to keyboard shortcuts.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference (object variable only) instead of a string.

This command can be applied to the following types of objects:

- Button, Default button, 3D button, Invisible button, Highlight button
- Radio button, 3D radio button, Picture button
- Check Box, 3D Check Box
- Pop-up menu, Drop-down List, Combo Box, Menu/Drop-down List
- Thermometer, Ruler

# OBJECT Get enterable

OBJECT Get enterable ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| Function result | Boolean | ⇐ | True = object(s) enterable; Otherwise, false |

## Description

The **OBJECT Get enterable** command returns True if the object or group of objects designated by *object* has the **enterable** attribute; otherwise, it returns False.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

# OBJECT GET EVENTS

OBJECT GET EVENTS ( {* ;} object ; arrEvents )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name or "" to designate the form (if * is specified) or<br>Field or variable (if * is omitted) |
| arrEvents | Longint array | ⇐ | Array of enabled events |

## Description

The **OBJECT GET EVENTS** command gets the current configuration of the form events for the object(s) designated by the *object* and * parameters.

Form events can be enabled/disabled either using the Property List, or using the **OBJECT SET EVENTS** command if it is called in the current process.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).
To get the configuration of events for the form itself, pass the optional * parameter and an empty string "" in object: in this case, you designate the current form.

**Note:** If you want to get events for a subform related to a table, you can only use the syntax based on the object name.

Pass a longint array in the *arrEvents* parameter. When the command is executed, this array is automatically sized and receives all the predefined or custom form events that are enabled for the object or the form. You can compare the values received with the constants of the "**Form Events**" theme.

Note that the *arrEvents* array is returned empty if no object method is associated with the object or if no form method is associated with the form.

## Example

You want to enable two events and get the list of events for an object:

```
ARRAY LONGINT($ArrCurrentEvents;0)
ARRAY LONGINT($ArrEnabled;2)
$ArrEnabled{1}:=On Header Click
$ArrEnabled{2}:=On Footer Click
OBJECT SET EVENTS(*;"Col1";$ArrEnabled;Enable events others unchanged)
OBJECT GET EVENTS(*;"Col1";$ArrCurrentEvents)
```

# OBJECT Get filter

OBJECT Get filter ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| Function result | Text | ⇐ | Name of filter |

## Description

The **OBJECT Get filter** command returns the name of any filter associated with the object or group of objects designated by *object*.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

# OBJECT Get focus rectangle invisible

OBJECT Get focus rectangle invisible ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable or field |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable or field (if * is omitted) |
| Function result | Longint | ⇐ | True = focus rectangle hidden,<br>False = focus rectangle shown |

## Description

The **OBJECT Get focus rectangle invisible** command returns the status of the visibility option for the focus rectangle of the object(s) designated by the *object* and * parameters for the current process . This setting corresponds to the **Hide focus rectangle** option that is available for enterable objects in the Property List in the Design mode. This command returns the current status of the option, as it is defined in Design mode or using the **OBJECT SET FOCUS RECTANGLE INVISIBLE** command.

**Note:** You can only use this option under Mac OS. It has no effect under Windows.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable or a field. In this case, you pass a variable reference instead of a string.

The command returns **True** when the focus rectangle is hidden and **False** when it is shown.

# OBJECT Get font

OBJECT Get font ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| Function result | Text | ⇐ | Name of font |

## Description

The **OBJECT Get font** command returns the name of the character font used by the form object(s) designated by *object*.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

# OBJECT Get font size

OBJECT Get font size ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| Function result | Longint | ⇐ | Size of font in points |

## Description

The **OBJECT Get font size** command returns the size (in points) of the character font used by the form object(s) designated by *object*.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

# OBJECT Get font style

OBJECT Get font style ( * ; object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| Function result | Longint | ⇐ | Font style |

## Description

The **OBJECT Get font style** command returns the current style of the character font used by the form object(s) designated by *object*.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

You can compare the value returned with the value of one or more of the following predefined constants, placed in the "**Font Styles**" theme:

| Constant | Type | Value |
|---|---|---|
| Plain | Longint | 0 |
| Bold | Longint | 1 |
| Italic | Longint | 2 |
| Underline | Longint | 4 |

# OBJECT Get format

OBJECT Get format ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a field or a variable |
| object | Form object | ➡ | Object name (if * is specified) or Field or variable (if * is omitted) |
| Function result | String | ⮌ | Object display format |

## Description

The **OBJECT Get format** command returns the current display format applied to the object specified in the *object* parameter.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (in this case, pass a string in *object*). If you do not pass this parameter, you indicate that the *object* parameter is a field or variable. In this case, you do not pass a string, but a field or variable reference.

This command returns the current display format of the object; in other words, the format as defined in the Design environment or using the **OBJECT SET FORMAT** command. **OBJECT Get format** works with all types of form objects (fields or variables) that accept a display format: Boolean, date, time, picture, string, number, as well as button grids, dials, thermometers, rulers, picture pop-up menus, picture buttons, 3D buttons, and list box headers. For more information on the display formats of these objects, refer to the documentation for the **OBJECT SET FORMAT** command.

**Note:** If you apply the command to a set of objects, the form of the last object selected is returned.

When the **OBJECT Get format** command is applied to date, time or picture objects (formats defined as constants), the string returned corresponds to the character code of the constant. To obtain the value of the constant, simply apply the **Character code** function to the result (see below).

## Example 1

This example allows you to obtain the value of the format constant applied to the picture variable named “myphoto”:

```
C_STRING(2;$format)
OBJECT SET FORMAT(*;"myphoto";Char(On background))
 `Apply background format (value = 3)
$format:=OBJECT Get format(*;"myphoto")
ALERT("Format number:"+String(Character code($format)))
 `Display value "3"
```

## Example 2

This example allows you to obtain the format applied to the Boolean field [Members]Marital_status:

```
C_STRING(30;$format)
$format:=OBJECT Get format([Members]Marital_status)
ALERT($format) `Display format, for example "Married;Single"
```

# OBJECT Get help tip

OBJECT Get help tip ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| Function result | Text | ⇐ | Help message of object |

## Description

The **OBJECT Get help tip** command returns the help message associated with the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

The command returns the current help message associated with the object, as it is defined in Design mode or for the process using the **OBJECT SET HELP TIP** command. The string returned shows the message as it appears when the form is executed. If it contains variable items (xliff *resname* or 4D references), they are interpreted according to the context.

## Example

The title of a picture button is stored as a help message. This title is stored in an xliff file and differs according to the current language of the application:

```
OBJECT SET HELP TIP(*;"button1";":xliff:btn_discover")
$helpmessage:=OBJECT Get help tip(*;"button1")
 // $helpmessage contains for example "Découvrir" with a French 4D and "Discover" with an English 4D.
```

# OBJECT Get horizontal alignment

OBJECT Get horizontal alignment ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an Object name (String) If omitted, object is a field or a variable |
| object | Form object | ⇒ | Object name (if * specified), or Field or variable (if * omitted) |
| Function result | Longint | ⇐ | Alignment code |

## Description

The **OBJECT Get horizontal alignment** command returns a code indicating the type of horizontal alignment applied to the object designated by the *object* and * parameters.

If you specify the optional * parameter, you indicate an object name (a string) in the *object* parameter. If you omit the * parameter, you indicate a field or variable in the *object* parameter. In this case, you specify a field or variable reference (field or variable objects only) instead of a string.

**Note:** If you apply the command to a group of objects, only the alignment value of the last object is returned.

The returned code corresponds to one of the following constants located in the **Form Objects (Properties)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Align center | Longint | 3 | |
| Align default | Longint | 1 | |
| Align left | Longint | 2 | |
| Align right | Longint | 4 | |
| wk justify | Longint | 5 | Available for 4D Write Pro areas only |

The form objects to which alignment can be applied are as follows:

- Scrollable areas
- Combo boxes
- Static text
- Group areas
- Pop up menu/Drop-down lists
- Fields
- Variables
- List boxes
- List box columns
- List box headers
- List box footers
- **4D Write Pro Reference** areas

# OBJECT Get indicator type

OBJECT Get indicator type ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | Longint | ⇐ | Indicator type |

## Description

The **OBJECT Get indicator type** command returns the current indicator type assigned to the thermometer(s) designated by the *object* and * parameters.

You can set the indicator type using the Property List in Design mode, or using the **OBJECT SET INDICATOR TYPE** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

You can compare the value returned by the command with the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Asynchronous progress bar | Longint | 3 | Circular indicator displaying continuous animation |
| Barber shop | Longint | 2 | Bar displaying continuous animation |
| Progress bar | Longint | 1 | Standard progress bar |

# OBJECT Get keyboard layout

OBJECT Get keyboard layout ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable or field |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable or field (if * is omitted) |
| Function result | String | ⇐ | Language code of layout, "" = no layout |

## Description

The **OBJECT Get keyboard layout** command returns the current keyboard layout associated with the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable or a field. In this case, you pass a reference instead of a name.

The command returns a string indicating the language code used, based on RFC3066, ISO639 and ISO3166. For more information, refer to the description of the **SET DATABASE LOCALIZATION** command.

# OBJECT Get list name

OBJECT Get list name ( {* ;} object {; listType} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| listType | Longint | ⇒ | Type of list: Choice list, Required list or Excluded list |
| Function result | Text | ⇐ | Name of list (specified in Design mode) |

## Description

The **OBJECT Get list name** command returns the name of the choice list associated with the object or group of objects designated by *object*. 4D lets you associate a choice list (created with the choice list editor in Design mode) with form objects using the form editor or the **OBJECT SET LIST BY NAME** command.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

You can use the optional *listType* parameter to designate the type of list that you want to get. By default, if you omit this parameter, the command returns the name of the choice list (list of values) associated with the object. You can also get the names of required lists or excluded lists by passing, in *listType*, one of the following constants found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Choice list | Longint | 0 | Simple list of values to choose from ("Choice List" option in the Property List) (default) |
| Excluded list | Longint | 2 | Lists values not accepted for entry ("Excluded List" option in the Property List) |
| Required list | Longint | 1 | Lists only values accepted for entry ("Required List" option in the Property List) |

If there is no list of the type defined associated with the *object*, the command returns an empty string ("").

# OBJECT Get list reference

OBJECT Get list reference ( {* ;} object {; listType} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| listType | Longint | ⇒ | Type of list: Choice list, Required list or Excluded list |
| Function result | ListRef | ⇐ | List reference number |

## Description

The **OBJECT Get list reference** command returns the reference number (ListRef) of the hierarchical list associated with the object or group of objects designated by *object* and *.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, this indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

By default, if you omit the *listType* parameter, the command returns the name of the choice list (list of values) associated with the object. You can also get the reference number for required lists or excluded lists by passing, in *listType*, one of the following constants found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Choice list | Longint | 0 | Simple list of values to choose from ("Choice List" option in the Property List) (default) |
| Excluded list | Longint | 2 | Lists values not accepted for entry ("Excluded List" option in the Property List) |
| Required list | Longint | 1 | Lists only values accepted for entry ("Required List" option in the Property List) |

If there is no hierarchical list associated with the object for the *listType* defined, the command returns 0.

# OBJECT GET MAXIMUM VALUE

OBJECT GET MAXIMUM VALUE ( {* ;} object ; maxValue )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| maxValue | Date, Time, Number | ⇐ | Current maximum value for object |

## Description

The **OBJECT GET MAXIMUM VALUE** command returns, in the *maxValue* variable, the current maximum value of the object(s) designated by the *object* and * parameters.

You can set the "Maximum Value" property using the Property List in Design mode, or using the **OBJECT SET MAXIMUM VALUE** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

# OBJECT GET MINIMUM VALUE

OBJECT GET MINIMUM VALUE ( {* ;} object ; minValue )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| minValue | Date, Time, Number | ⇐ | Current minimum value for object |

## Description

The **OBJECT GET MINIMUM VALUE** command returns, in the *minValue* variable, the current minimum value of the object(s) designated by the *object* and * parameters.

You can set the "Minimum Value" property using the Property List in Design mode, or using the **OBJECT SET MINIMUM VALUE** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

# OBJECT Get multiline

OBJECT Get multiline ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Longint | ⇐ | Multiline status of object |

## Description

The **OBJECT Get multiline** command returns the current state of the "Multiline" option for the object(s) designated by the *object* and * parameters.

You can set the "Multiline" option for an object using the Property List, or using the **OBJECT SET MULTILINE** command option.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

The value returned corresponds to one of the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Multiline Auto | Longint | 0 | In single-line areas, words located at the end of lines are truncated and there are no line returns.<br>In multiline areas, 4D carries out automatic line returns. |
| Multiline No | Longint | 2 | There are never line returns: the text is always displayed on a single row. If the Alpha or Text field or variable contains carriage returns, the text located after the first carriage return is removed as soon as the area is modified. |
| Multiline Yes | Longint | 1 | In single-line areas, the text is displayed up to the first carriage return or until the last word that can be displayed entirely. 4D inserts line returns; it is possible to scroll the contents of the area by pressing the down arrow key.<br>In multiline areas, 4D carries out automatic line returns. |

**Note:** If you apply the **OBJECT Get multiline** command to an object that does not support the "Multiline" option, the command returns 0.

# OBJECT Get name

OBJECT Get name {( selector )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| selector | Longint | ➡ | Object category |
| Function result | Text | ⮌ | Name of object |

## Description

The **OBJECT Get name** command returns the name of a form object.

The command can be used to designate two types of objects according to the value of the *selector* parameter. In this parameter, you can pass one of the following constants (placed in the "**Form Objects (Access)**" theme:

- Object current or *selector* omitted: If you pass this selector or omit the *selector* parameter, the command returns the name of the object from which it was called (object method or submethod called by the object method). In this case, the command must be called in the context of a form object, otherwise it returns an empty string.
- Object with focus: If you pass this selector, the command returns the name of the object that has the focus in the form.

## Example

Object method for "bValidateForm" button:

```
$btnName:=OBJECT Get name(Object current)
```

After the execution of this object method, the *$btnName* variable contains the "bValidateForm" value.

# OBJECT Get placeholder

OBJECT Get placeholder ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Text | ⇐ | Placeholder text associated with object |

## Description

The **OBJECT Get placeholder** command returns the placeholder text associated with the object(s) designated by the *object* and * parameters. If there is no placeholder text associated with the object, the command returns an empty string.

You can define the placeholder text either using the Property List, or using the **OBJECT SET PLACEHOLDER** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

If the placeholder is an xliff reference defined using the Property List, the command returns the original reference in the form ":xliff:*resname*", and not its calculated value.

## Example

You want to get the field placeholder text:

```
$txt:=OBJECT Get placeholder([People]LastName)
```

# OBJECT Get pointer

OBJECT Get pointer ( {selector }{;}{ objectName {; subformName}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| selector | Longint | ⇒ | Object category |
| objectName | Text | ⇒ | Object name |
| subformName | Text | ⇒ | Subform object name |
| Function result | Pointer | ⇐ | Pointer to object variable |

## Description

The **OBJECT Get pointer** command returns a pointer to the variable of a form object.

This command can be used to designate different objects according to the value of the *selector* parameter. You can pass one of the following constants (found in the "**Form Objects (Access)**" theme) in this parameter:

- Object current or *selector* omitted: If you omit the *selector* parameter or pass this selector, the command returns a pointer to the variable associated with the current object (object whose method is executing).
  **Note:** This is strictly equivalent to the previous functioning of the **Self** command. This command is only kept for compatibility reasons.
- Object with focus: If you pass this selector, the command returns a pointer to the variable associated with the object that has the focus in the form. The last two optional parameters are ignored if they are passed.
  **Note:** This is strictly equivalent to the functioning of the **Focus object** command. This command is now obsolete beginning with 4D v12.
- Object subform container: If you pass this selector, the command returns a pointer to the variable bound with the subform container. The last two optional parameters are ignored if they are passed. This selector can therefore only be used in the context of a form used as a subform, so as to access the variable bound with the container object.
- Object named: If you pass this selector, you must also pass the second parameter, *objectName*. In this case, the command returns a pointer to the variable associated with the object whose name was passed in this parameter.
  **Note:** If *objectName* corresponds to a subform and the "Output subform" option is checked, the command returns a pointer to the table of the subform if a source table is specified; otherwise it returns Nil.

The optional *subformName* parameter lets you retrieve a pointer to an *objectName* object that does not belong to the current context, in other words, in the parent form. To be able to use this parameter, you must have passed the Object named selector.
When the *subformName* parameter is passed, the **OBJECT Get pointer** command first looks for the subform object named *subformName* in the current form, then looks inside this subform for an object named *objectName*. If this object is found, it returns a pointer to the variable of this object.

## Example

Given a form "SF" used twice as a subform in the same parent form. The subform objects are named "SF1" and "SF2". The "SF" form contains an object named *CurrentValue*. In the "On Load" form event of the form method of the parent form, we want to initialize the *CurrentValue* object of SF1 to "January" and that of SF2 to "February":

```
C_POINTER($Ptr)
$Ptr:=OBJECT Get pointer(Object named;"CurrentValue";"SF1")
$Ptr->:="January"
$Ptr:=OBJECT Get pointer(Object named;"CurrentValue";"SF2")
$Ptr->:="February"
```

# OBJECT GET PRINT VARIABLE FRAME

OBJECT GET PRINT VARIABLE FRAME ( {* ;} object ; variableFrame {; fixedSubform} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| variableFrame | Boolean | ⇐ | True = Variable frame printing, False = Fixed frame printing |
| fixedSubform | Longint | ⇐ | Option for printing subforms in fixed size |

## Description

The **OBJECT GET PRINT VARIABLE FRAME** command gets the current configuration of the variable frame print options for the object(s) designated by the *object* and * parameters.

Variable frame printing properties can be defined using the Property List, or using the **OBJECT SET PRINT VARIABLE FRAME** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, this indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In the *variableFrame* parameter, the command returns a Boolean variable whose value indicates the enabled (**True**) or disabled (**False**) state of variable frame printing.

If the *object* is a subform and if variable frame printing is disabled (**False**), the command also returns, in the *fixedSubform* parameter, the fixed frame print option of the subform. You can compare the value returned with the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Print Frame fixed with multiple records | Longint | 2 | The frame remains the same size, but 4D prints the form several times to include all the records. |
| Print Frame fixed with truncation | Longint | 1 | 4D prints only the records that fit into the area of the subform. The form is printed only once and those records that are not printed are ignored. |

# OBJECT GET RESIZING OPTIONS

OBJECT GET RESIZING OPTIONS ( {* ;} object ; horizontal ; vertical )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| horizontal | Longint | ⇐ | Horizontal resizing option |
| vertical | Longint | ⇐ | Vertical resizing option |

## Description

The **OBJECT GET RESIZING OPTIONS** command returns the current resizing options for the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

The command returns the current resizing options, as set in the Design mode or for the process using the **OBJECT SET RESIZING OPTIONS** command. These options specify the display of the object when the form window is resized.

The *horizontal* parameter returns a value indicating the horizontal resizing option that is set for the object. You can compare the value received with the following constants, found in the **Form Objects (Properties)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Resize horizontal grow | Longint | 1 | If the window grows by 50% in width, the object is expanded by 50% to the right. |
| Resize horizontal move | Longint | 2 | If the window grows by 100 pixels in width, the object is moved 100 pixels to the right. |
| Resize horizontal none | Longint | 0 | If the window is expanded in width, neither the width nor the position of the object changes. |

The *vertical* parameter returns a value indicating the vertical resizing option that is set for the object. You can compare the value received with the following constants, found in the **Form Objects (Properties)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Resize vertical grow | Longint | 1 | If the window grows by 50% in height, the object is lengthened by 50% towards the bottom. |
| Resize vertical move | Longint | 2 | If the window grows by 100 pixels in height, the object is moved 100 pixels towards the bottom. |
| Resize vertical none | Longint | 0 | If the window is expanded in height, neither the height nor the position of the object changes. |

# OBJECT GET RGB COLORS

OBJECT GET RGB COLORS ( {* ;} object ; foregroundColor {; backgroundColor {; altBackgrndColor}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| foregroundColor | Longint | ⇐ | RGB color value for foreground |
| backgroundColor | Longint | ⇐ | RGB color value for background |
| altBackgrndColor | Longint | ⇐ | RGB color value for alternating background |

## Description

The **OBJECT GET RGB COLORS** command returns the foreground and background colors of the object or group of objects designated by *object*.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

When the command is applied to a List box type object, the alternating background color for the rows can be returned in the *altBackgrndColor* parameter. In this case, the value of *backgroundColor* is used for the background of odd-numbered rows only.

The RGB color values returned in the *foregroundColor*, *backgroundColor* and *altBackgrndColor* parameters can be 4-byte longints of the format (0x00RRGGBB) or negative values corresponding to the "system" colors. In the latter case, you can compare the value obtained with the constants of the **SET RGB COLORS** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Background color | Longint | -2 | |
| Background color none | Longint | -16 | This constant can only be used with the *backgroundColor* and *altBackgrndColor* parameters. |
| Dark shadow color | Longint | -3 | |
| Disable highlight item color | Longint | -11 | |
| Foreground color | Longint | -1 | |
| Highlight menu background color | Longint | -9 | |
| Highlight menu text color | Longint | -10 | |
| Highlight text background color | Longint | -7 | |
| Highlight text color | Longint | -8 | |
| Light shadow color | Longint | -4 | |

**Note:** The "system" colors are applied using the **OBJECT SET RGB COLORS** command.

For more information about the format of the *foregroundColor*, *backgroundColor* and *altBackgrndColor* parameters, refer to the description of the **OBJECT SET RGB COLORS** command.

# OBJECT GET SCROLL POSITION

OBJECT GET SCROLL POSITION ( {* ;} object ; vPosition {; hPosition} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field or table (if * is omitted) |
| vPosition | Longint | ⇐ | Number of first line displayed or Vertical scrolling in pixels (pictures) |
| hPosition | Longint | ⇐ | Number of first column displayed or Horizonal scrolling in pixels (pictures) |

## Description

The **OBJECT GET SCROLL POSITION** returns, in the *vPosition* and *hPosition* parameters, information related to the position of the scroll bars of the form object designated by the * and *object* parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is the name of an object of the subform, hierarchical list, scrollable area, list box or picture type (in this case, pass a string in object). If you do not pass this parameter, you indicate that the *object* parameter is a variable (*ListRef* of hierarchical list, picture or list box variable) or a field.

**Note:** With subform type objects, only the syntax using an * is supported.

If *object* designates a list type object (subform, list form, hierarchical list, scrollable area or list box), *vPosition* returns the number of the first line displayed in the object. *hPosition* (list box only) returns the number of the first column that is completely visible in the left part of the list box. With other types of objects, this parameter returns 0.

If *object* designates a picture (variable or field), *vPosition* returns the vertical movement and *hPosition* the horizontal movement of the picture. These values are expressed in pixels with respect to the origin of the picture in the local coordinate system.

# OBJECT GET SCROLLBAR

OBJECT GET SCROLLBAR ( {* ;} object ; horizontal ; vertical )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| horizontal | Boolean, Longint | ⇐ | Visibility of horizontal scrollbar |
| vertical | Boolean, Longint | ⇐ | Visibility of vertical scrollbar |

## Description

The **OBJECT GET SCROLLBAR** command is used to find out the displayed/hidden status of the horizontal and vertical scrollbars of the object or group of objects designated by *object*.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

In the *horizontal* and *vertical* parameters, you can pass Boolean or longint type variables:

- When you pass Boolean variables, the value returned reflects the **current** state of the scrollbar:
  - If the scrollbar has been defined as hidden, the parameter receives False,
  - If the scrollbar has been defined as displayed, the parameter receives True,
  - If the scrollbar has been set to automatic mode, the parameter receives either True or False depending on the current display state of the object.
- When you pass longint variables, the value returned reflects the visibility defined for the scrollbar:
  - If the scrollbar has been defined as hidden, the parameter receives 0,
  - If the scrollbar has been defined as displayed, the parameter receives 1,
  - If the scrollbar has been set to automatic mode, the parameter receives 2.

This command can be used with the following form objects:

- Picture or text object fields and variables
- List boxes,
- Hierarchical lists,
- Subforms.

For more information, refer to the description of the **OBJECT SET SCROLLBAR** command.

# OBJECT GET SHORTCUT

OBJECT GET SHORTCUT ( {* ;} object ; key ; modifiers )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| key | String | ⇐ | Key associated with object |
| modifiers | Longint | ⇐ | Modifier key mask or combination of masks |

## Description

The **OBJECT GET SHORTCUT** command returns the keyboard shortcut associated with the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

The *key* parameter returns the character associated with the key (in the case of a standard key) or a string enclosed in brackets designating the key (in the case of a function key). You can compare this value with the constants of the **Shortcut and Associated Keys** theme (see the **OBJECT SET SHORTCUT** command).

The *modifiers* parameter returns a value indicating the modifier key(s) associated with the shortcut. If there are several modifier keys combined, the command returns the sum of their values. You can compare the value returned with the following constants of the **Events (Modifiers)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Command key mask | Longint | 256 | Ctrl key under Windows, Command key under OS X |
| Control key mask | Longint | 4096 | Ctrl key under OS X, or right click under Windows and OS X |
| Option key mask | Longint | 2048 | Alt key (also called Option under OS X) |
| Shift key mask | Longint | 512 | Windows and OS X |

If there are no modifier keys for the shortcut, *modifiers* returns 0.

**Note:** If the *object* parameter designates several objects in the form that have different settings, the command returns "" in *key* and 0 in *modifiers*.

# OBJECT Get style sheet

OBJECT Get style sheet ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Text | ⇐ | Name of style sheet |

## Description

The **OBJECT Get style sheet** command returns the name of the style sheet associated with the object(s) designated by the *object* and * parameters.

Style sheets may have been assigned in Design mode using the Property List, or for the current process using the **OBJECT SET STYLE SHEET** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

The command can return either:

- a style sheet name,
- an empty string ("") if no style sheet is assigned, or
- if an automatic style sheet is assigned, one of the following constants found in the "**Font Styles**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Automatic style sheet | String | __automatic__ | Used by default for all objects |
| Automatic style sheet_additional | String | __automatic_additional_text__ | Supported by static text, fields and variables only. Used for additional text in dialog boxes. |
| Automatic style sheet_main | String | __automatic_main_text__ | Supported by static text, fields and variables only. Used for main text in dialog boxes. |

If the command designates several objects, the style sheet returned is only meaningful if the style sheet is assigned to all of the objects.

# OBJECT GET SUBFORM

OBJECT GET SUBFORM ( {* ;} object ; tablePtr ; detailSubform {; listSubform} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| tablePtr | Table | ⇐ | Pointer to table of form |
| detailSubform | Text | ⇐ | Name of detail form of subform |
| listSubform | Text | ⇐ | Name of list form of subform (table form) |

## Description

The **OBJECT GET SUBFORM** command gets the name(s) of the form(s) associated with the subform object designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

In the *tablePtr* parameter, the command returns a pointer to the table of the form(s) used. If the subform uses a project form, this parameter contains **Nil**.
In the *detailSubform* parameter, the command returns the name of the detail form used in the subform.
In the *listSubform* parameter, the command returns the name of the list form used in the subform. If there is no list form, an empty string is returned.

# OBJECT GET SUBFORM CONTAINER SIZE

OBJECT GET SUBFORM CONTAINER SIZE ( width ; height )

| Parameter | Type | | Description |
|---|---|---|---|
| width | Longint | ⇦ | Width of subform object |
| height | Longint | ⇦ | Height of subform object |

## Description

The **OBJECT GET SUBFORM CONTAINER SIZE** command returns the *width* and *height* (in pixels) of a "current" subform object, displayed in the parent form.

This command must be called from the method of a form used as a subform and displayed in a subform object. It returns the *width* and *height* of the object containing the subform.

This command is useful, for example, in the case where subform objects must be resized according to the characteristics of the subform object itself. In the On Load form event, the subform can call this command to calculate the space at its disposal in order to display its contents.

**Note:** It is not possible to use the On Resize event directly in the subform method. Since this event is linked to the resizing of a window, it is generated only in the method of the parent form. You can, however, explicitly call the subform from this event of the parent form using, for instance, the **EXECUTE METHOD IN SUBFORM** command.

- If the command is called from a form that is not being used as a subform, it returns the current size of the form window.
- If the command is called outside of the context of screen display (for example, during form printing), it returns 0 in *width* and *height*.

# OBJECT Get text orientation

OBJECT Get text orientation ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Longint | ⇐ | Angle of text rotation |

## Description

The **OBJECT Get text orientation** command returns the current orientation value applied to the text of the object(s) designated by the *object* and * parameters.

You can set the "Orientation" option for an object in Design mode using the Property List, or using the **OBJECT SET TEXT ORIENTATION** command.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

The value returns corresponds to one of the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Orientation 0° | Longint | 0 | No rotation (default value) |
| Orientation 180° | Longint | 180 | Orientation of text to 180° clockwise |
| Orientation 90° left | Longint | 270 | Orientation of text to 90° counter-clockwise |
| Orientation 90° right | Longint | 90 | Orientation of text to 90° clockwise |

## Example

Given the following object (where a "90° left" orientation was applied in the Form editor):


When the form is executed, if you call the following statement:

```
OBJECT SET TEXT ORIENTATION(*;"myText";Orientation 180° )
```

... then the object appears as follows:


```
$vOrt:=OBJECT Get text orientation(*;"myText") //$vOrt=180
```

# OBJECT Get three states checkbox

OBJECT Get three states checkbox ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Boolean | ⇐ | True = three-states checkbox, False = standard checkbox |

## Description

The **OBJECT Get three states checkbox** command returns the current state of the "Three-States" property for the checkbox(es) designated by the *object* and * parameters.

You can set the "Three-States" property either using the Property List, or using the **OBJECT SET THREE STATES CHECKBOX** command if it was called in the current process.

# OBJECT Get title

OBJECT Get title ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| Function result | Text | ⇐ | Title of button |

## Description

The **OBJECT Get title** command returns the title (label) of the form object(s) designated by *object*. You can use this command with all types of simple objects that contain a label:

- buttons,
- check boxes
- radio buttons
- static texts
- group areas.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

# OBJECT Get type

OBJECT Get type ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | Longint | ⇐ | Type of object |

## Description

The **OBJECT Get type** command returns the type of the object designated by the *object* and * parameters in the current form.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). This syntax is mandatory if you are processing static objects such as lines or rectangles.
If you do not pass this parameter, it indicates that the *object* parameter is a variable. In this case, you pass a variable reference instead of a string.

**Note:** If you apply this command to a set of objects, the type of the last object is returned.

The value returned corresponds to one of the following constants, available in the "**Form Object Types**" theme:

| Constant | Type | Value |
|---|---|---|
| Object type 3D button | Longint | 16 |
| Object type 3D checkbox | Longint | 26 |
| Object type 3D radio button | Longint | 23 |
| Object type button grid | Longint | 20 |
| Object type checkbox | Longint | 25 |
| Object type combobox | Longint | 11 |
| Object type dial | Longint | 28 |
| Object type group | Longint | 21 |
| Object type groupbox | Longint | 30 |
| Object type hierarchical list | Longint | 6 |
| Object type hierarchical popup menu | Longint | 13 |
| Object type highlight button | Longint | 17 |
| Object type invisible button | Longint | 18 |
| Object type line | Longint | 32 |
| Object type listbox | Longint | 7 |
| Object type listbox column | Longint | 9 |
| Object type listbox footer | Longint | 10 |
| Object type listbox header | Longint | 8 |
| Object type matrix | Longint | 35 |
| Object type oval | Longint | 34 |
| Object type picture button | Longint | 19 |
| Object type picture input | Longint | 4 |
| Object type picture popup menu | Longint | 14 |
| Object type picture radio button | Longint | 24 |
| Object type plugin area | Longint | 38 |
| Object type popup dropdown list | Longint | 12 |
| Object type progress indicator | Longint | 27 |
| Object type push button | Longint | 15 |
| Object type radio button | Longint | 22 |
| Object type radio button field | Longint | 5 |
| Object type rectangle | Longint | 31 |
| Object type rounded rectangle | Longint | 33 |
| Object type ruler | Longint | 29 |
| Object type splitter | Longint | 36 |
| Object type static picture | Longint | 2 |
| Object type static text | Longint | 1 |
| Object type subform | Longint | 39 |
| Object type tab control | Longint | 37 |
| Object type text input | Longint | 3 |
| Object type unknown | Longint | 0 |
| Object type web area | Longint | 40 |
| Object type write pro area | Longint | 41 |

## Example

You want to load a form and get a list of all the objects of list boxes that it contains.

```
FORM LOAD("MyForm")
ARRAY TEXT(arrObjects;0)
FORM GET OBJECTS(arrObjects)
ARRAY LONGINT(ar_type;Size of array(arrObjects))
For($i;1;Size of array(arrObjects))
```

```
      ar_type{$i}:=OBJECT Get type(*;arrObjects{$i})
      If(ar_type{$i}=Object type listbox)
         ARRAY TEXT(arrLBObjects;0)
         LISTBOX GET OBJECTS(*;arrObjects{$i};arrLBObjects)
      End if
   End for
   FORM UNLOAD
```

# OBJECT Get vertical alignment

OBJECT Get vertical alignment ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| Function result | Longint | ⇐ | Type of alignment |

## Description

The **OBJECT Get vertical alignment** command returns a value indicating the type of vertical alignment applied to the object designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

**Note:** If you apply this command to a set of objects, only the alignment value for the last object is returned.

The value returned corresponds to one of the following constants, found in the **Form Objects (Properties)** theme:

| Constant | Type | Value |
|---|---|---|
| Align bottom | Longint | 4 |
| Align center | Longint | 3 |
| Align top | Longint | 2 |

Vertical alignment can be applied to the following types of form objects:

- list boxes,
- list box columns,
- list box headers and footers.

# OBJECT Get visible

OBJECT Get visible ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| Function result | Boolean | ⇐ | True = object(s) visible; Otherwise, False |

## Description

The **OBJECT Get visible** command returns True if the object or group of objects designated by *object* has the visible attribute and False otherwise.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

# OBJECT Is styled text

OBJECT Is styled text ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| Function result | Boolean | ⇐ | True if object is a multi-style text, False otherwise |

## Description

The **OBJECT Is styled text** command returns **True** when the "Multi-style" option is checked for the object(s) designated by the *object* and * parameters.

The "Multi-style" option lets you use rich test areas including individual style variations. For more information, refer to **GET DATA SOURCE LIST** in the *Design Reference* manual.

Multi-style objects can be managed by programming using the commands of the "**Styled Text**" theme.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

**Note:** The **OBJECT Is styled text** command returns **True** when it is applied to a 4D Write Pro area.

## Example

A form contains a field represented by two different objects; one of the objects has the "Multi-style" property checked, and the other one does not. You can write:

```
$Style:=OBJECT Is styled text(*;"Styled_text")
 // returns True ("Multi-style" option is checked)

$Style:=OBJECT Is styled text(*;"Plain_text")
 // returns False ("Multi-style" option is not checked)
```

# OBJECT MOVE

OBJECT MOVE ( {* ;} object ; moveH ; moveV {; resizeH {; resizeV {; *}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified= object is an object name (string) If omitted = object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| moveH | Longint | ⇒ | Value of the horizontal move of the object (>0 = to the right, <0 = to the left) |
| moveV | Longint | ⇒ | Value of the vertical move of the object (>0 = to the bottom, <0 = to the top) |
| resizeH | Longint | ⇒ | Value of the horizontal resize of the object |
| resizeV | Longint | ⇒ | Value of the vertical resize of the object |
| * | Operator | ⇒ | If specified = absolute coordinates If omitted = relative coordinates |

## Description

The **OBJECT MOVE** command allows you to move the object(s) in the current form, defined by the * and *object* parameters *moveH* pixels horizontally and *moveV* pixels vertically.

It is also possible (optionally) to resize the object(s) *resizeH* pixels horizontally and *resizeV* pixels vertically.

The direction to move and resize depend on the values passed to the *moveH* and *moveV* parameters:

- If the value is positive, objects are moved and resized to the right and to the bottom, respectively.
- If the value is negative, objects are moved and resized to the left and to the top, respectively.

If you pass the first optional parameter *, you indicate that the *object* parameter is a parameter name (a string of characters). If you don't pass the * parameter, *object* is a field or a variable. In this case, you don't pass a string but a field or variable reference (only a field or variable of type object).

If you pass an object name to object and use the wildcard character ("@") to select more than one object, all the objects concerned will be moved or resized.

**Note:** Since 4D version 6.5, it is possible to set the interpretation mode of the wildcard character ("@"), when it is included in a string of characters. This option has an impact on the "Object Properties" commands. Please refer to the 4D Design Mode manual.

By default, the values *moveH*, *moveV*, *resizeH* and *resizeV* modify the coordinates of the object relative to its previous position. If you want the parameters to define the absolute parameters, pass the last optional parameter *.

This command works in the following contexts:

- Data entering in Input forms,
- Forms displayed using the **DIALOG** command,
- Headers and footers of Output forms displayed with **MODIFY SELECTION** or **DISPLAY SELECTION** commands,
- Form printing events.

## Example 1

The following statement moves "button_1" 10 pixels to the right, 20 pixels to the top and resizes it to 30 pixels in width and 40 in height:

```
OBJECT MOVE(*;"button_1";10;-20;30;40)
```

## Example 2

The following statement moves "button_1" to the following coordinates (10;20) (30;40):

```
OBJECT MOVE(*;"button_1";10;20;30;40;*)
```

# OBJECT SET ACTION

OBJECT SET ACTION ( {* ;} object ; action )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| action | Text | ⇒ | Action to associate |

## Description

The **OBJECT SET ACTION** command modifies, for the current process, the standard action associated with the object(s) designated by the *object* and * parameters.

**Note:** Standard actions can also be set using the Property list in the Design mode.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In the *action* parameter, pass an action code (string) indicating the standard action to associate with the object. You can pass one of the following constants, found in the "**Text Values for Associated Standard Action**" theme:

| Constant | Type | Value |
| --- | --- | --- |
| Object Accept action | String | 2 |
| Object Add subrecord action | String | 14 |
| Object Automatic splitter action | String | 16 |
| Object Cancel action | String | 1 |
| Object Clear action | String | 21 |
| Object Copy action | String | 19 |
| Object Cut action | String | 18 |
| Object Database Settings action | String | 32 |
| Object Delete record action | String | 7 |
| Object Delete subrecord action | String | 13 |
| Object Edit subrecord action | String | 12 |
| Object First page action | String | 10 |
| Object First record action | String | 5 |
| Object Goto page action | String | 15 |
| Object Last page action | String | 11 |
| Object Last record action | String | 6 |
| Object MSC action | String | 36 |
| Object Next page action | String | 8 |
| Object Next record action | String | 3 |
| Object No standard action | String | 0 |
| Object Open back URL action | String | 37 |
| Object Open next URL action | String | 38 |
| Object Paste action | String | 20 |
| Object Previous page action | String | 9 |
| Object Previous record action | String | 4 |
| Object Quit action | String | 27 |
| Object Redo action | String | 31 |
| Object Refresh current URL action | String | 39 |
| Object Return to Design mode action | String | 35 |
| Object Select all action | String | 22 |
| Object Show Clipboard action | String | 23 |
| Object Stop loading URL action | String | 40 |
| Object Test Application action | String | 26 |
| Object Undo action | String | 17 |

## Example

You want to associate the **Validate** standard action with a button:

```
OBJECT SET ACTION(*;"bValidate";Object Accept action)
```

# OBJECT SET AUTO SPELLCHECK

OBJECT SET AUTO SPELLCHECK ( {* ;} object ; autoSpellcheck )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable or field |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable or field (if * is omitted) |
| autoSpellcheck | Boolean | ⇒ | True = automatic spell-checking,<br>False= no automatic spell-checking |

## Description

The **OBJECT SET AUTO SPELLCHECK** command sets or dynamically modifies the status of the **Auto spellcheck** option for the object(s) designated by the *object* and * parameters for the current process. This option enables or disables the automatic spellcheck when data is entered for the object (Text type objects only).

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable or a field. In this case, you pass a reference instead of a name.

Pass **True** in *autoSpellcheck* to enable this function for the object, and **False** to disable it.

# OBJECT SET BORDER STYLE

OBJECT SET BORDER STYLE ( {* ;} object ; borderStyle )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| borderStyle | Longint | ⇒ | Border line style |

## Description

The **OBJECT SET BORDER STYLE** command modifies the border line style of the object(s) designated by the *object* and * parameters.

The "Border Line Style" property modifies the appearance of the object outlines. For more information, refer to **Border Line Style** in the *Design Reference* manual.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In the *borderStyle* parameter, pass the value of the border line style that you want to apply to the object. You can pass one of the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Border Dotted | Longint | 2 | Objects appear framed with a dotted 1-pt. border line |
| Border Double | Longint | 5 | Objects appear framed with a double line, i.e., two continuous 1-pt. lines separated by a pixel |
| Border None | Longint | 0 | Objects appear with no border |
| Border Plain | Longint | 1 | Objects appear framed with a continuous 1-pt. border line |
| Border Raised | Longint | 3 | Objects appear framed with a 3D effect (raised) |
| Border Sunken | Longint | 4 | Objects appear framed with a sunken 3D effect |
| Border System | Longint | 6 | The border line is drawn based on the graphic specifications of the system |

# OBJECT SET COLOR

OBJECT SET COLOR ( {* ;} object ; color {; altColor} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable |
| object | Field, Variable | ⇒ | Object Name (if * is specified), or Field or Variable (if * is omitted) |
| color | Longint | ⇒ | New colors for the object |
| altColor | Longint | ⇒ | Alternating colors for a list box |

## Description

The **OBJECT SET COLOR** command sets the foreground and background colors of the form objects specified by *object*. If *object* is a list box, an additional parameter is used to set the foreground and background colors for even-numbered rows (alternating colors).

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the **Object Properties** section.

The *color* parameter specifies both foreground and background colors. The color is calculated as:

**Color:=-(Foreground+(256 * Background))**

where **Foreground** and **Background** are color numbers (from 0 to 255) within the color palette.

**Color** is always a negative number. For example, if the foreground color is to be 20 and the background color is to be 10, then *color* is – (20 + (256 * 10)) or –2580.

*altColor* is used to specify an alternative background color for the even-numbered rows of a list box or a list box column. In *altColor*, you must pass only the "background" part of the color formula, i.e. **AltColor:=-(256 * Background)**.
When this parameter is passed, the *color* parameter will be applied only to odd-numbered rows. Using alternating colors makes lists easier to read. If *object* specifies the list box object, alternating colors are used throughout the entire list box. If *object* specifies a column, only the column will use the colors set.

**Note:** You can see the color palette in the Form Editor's Property List window.

The numbers of the commonly used colors are provided by the following predefined constants, located in the "**Colors**" theme:

| Constant | Type | Value |
|---|---|---|
| Black | Longint | 15 |
| Blue | Longint | 6 |
| Brown | Longint | 13 |
| Dark blue | Longint | 5 |
| Dark brown | Longint | 10 |
| Dark green | Longint | 9 |
| Dark grey | Longint | 11 |
| Green | Longint | 8 |
| Grey | Longint | 14 |
| Light blue | Longint | 7 |
| Light grey | Longint | 12 |
| Orange | Longint | 2 |
| Purple | Longint | 4 |
| Red | Longint | 3 |
| White | Longint | 0 |
| Yellow | Longint | 1 |

**Note:** While **OBJECT SET COLOR** works with indexed colors within the default 4D color palette, the **OBJECT SET RGB COLORS** command allows you to work with any RGB color. To reestablish automatic colors for an object, use the **OBJECT SET RGB COLORS** command with the Default foreground color and Default background color constants.

## Example 1

The following example sets the color of the text area shown below in the form editor:


After executing the following statement:

```
OBJECT SET COLOR(*;"Mytext";-(Yellow+(256*Red)))
```

... the area appears as follows:


## Example 2

You want to set an alternating background color for a column in the list box. You can write:

```
OBJECT SET COLOR(*;"countryCol";-(Dark blue+(256*Red));-(256*Orange))
```

# OBJECT SET CONTEXT MENU

OBJECT SET CONTEXT MENU ( {* ;} object ; contextMenu )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| contextMenu | Boolean | ⇒ | True = enable context menu, False = disable context menu |

## Description

The **OBJECT SET CONTEXT MENU** command enables or disables, for the current process, the association of a context menu by default with the object(s) designated by the *object* and * parameters.

The "Context Menu" option is available for text type entry areas, Web areas and pictures. You can use it to associate a standard action menu with these objects depending on their type (for example Copy/Paste for text objects). For more information, refer to the *Design Reference* manual.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

Pass **True** in the *contextMenu* parameter to enable the context menu, and **False** to disable it.

# OBJECT SET COORDINATES

OBJECT SET COORDINATES ( {* ;} object ; left ; top {; right ; bottom} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Longint | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| left | Longint | ⇒ | Left coordinate of object in pixels |
| top | Longint | ⇒ | Top coordinate of object in pixels |
| right | Longint | ⇒ | Right coordinate of object in pixels |
| bottom | Longint | ⇒ | Bottom coordinate of object in pixels |

## Description

The **OBJECT SET COORDINATES** command modifies the location and, optionally, the size of the object(s) designated by the *object* and * parameters for the current process.

**Note:** This command is the equivalent of using the **OBJECT MOVE** command and passing its 2nd * parameter.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In the *left* and *top* parameters, pass the new absolute coordinates of the object in the form. These coordinates must be expressed in pixels with respect to the top left corner of the form.

You can also pass absolute coordinate values in the *right* and *bottom* parameters, indicating the bottom right corner of the object. If this corner does not correspond to the corner of the object after application of the *left* and *top* parameters, the object is resized accordingly.

**Note:** If you want to move an object relative to its initial position, we recommend using the existing **OBJECT MOVE** command.

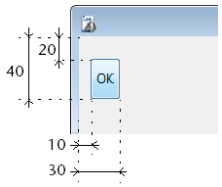
This command only functions in the following contexts:

- Input forms in entry mode,
- Forms displayed using the **DIALOG** command,
- Headers and footers of output forms displayed by the **MODIFY SELECTION** or **DISPLAY SELECTION** command,
- Forms being printed.

## Example

The following statement places the "button_1" object at the (10,20) (30,40) coordinates:

```
OBJECT SET COORDINATES(*;"button_1";10;20;30;40)
```



# OBJECT SET CORNER RADIUS

OBJECT SET CORNER RADIUS ( {* ;} object ; radius )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| radius | Longint | ⇒ | New radius of rounded corners (in pixels) |

## Description

The **OBJECT SET CORNER RADIUS** command modifies the radius of corners for the rounded rectangle object(s) whose name(s) you passed in the *object* parameter. The new radius is only set for the process and is not saved in the form itself.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

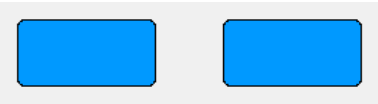
**Note:** In the current version of 4D, this command only applies to rounded rectangles (which are static objects). As a result, only the syntax based on the object name (using the * parameter) is supported.

In the *radius* parameter, you pass a new radius value in pixels to be applied to the corners of the object. By default, this value is 5 pixels.

**Note:** You can also modify this value at the form level using the Property list (see **Corner radius (rectangles)**).

## Example

You have the following rectangles in your form, named respectively "Rect1" and "Rect2":



You can execute the following code to change their corners:

```
OBJECT SET CORNER RADIUS(*;"Rect@";20)
```

# OBJECT SET DATA SOURCE

OBJECT SET DATA SOURCE ( {* ;} object ; dataSource )

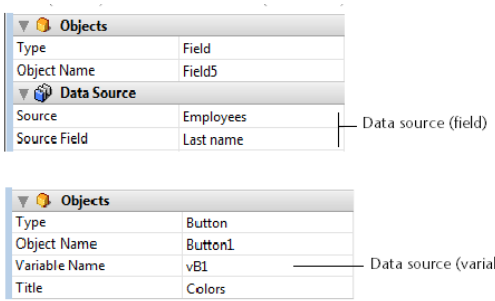
| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| dataSource | Pointer | ⇒ | Pointer to new data source for object |

## Description

The **OBJECT SET DATA SOURCE** command modifies the data source of the object(s) designated by the *object* and * parameters.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

The data source is the field or variable whose value is represented by the object when the form is executed. In Design mode, the data source is defined in the Property list, usually through the Source and Source Field (fields) or Variable Name (variables) row:




Except for list boxes (see below), all data sources of the form can be modified by this command. It is up to the developer to ensure the consistency of the changes made.

In the case of list boxes, the following points must be considered:

- Data source modifications must take the list box type into account: for example, it is not possible to use a field as the data source for a column in an array type list box.
- For selection type list boxes, it is not possible to modify or read the data source of the list box object itself: in this case, it is an internal reference and not a data source.
- This command is mainly used in the context of array type list boxes. For selection type list boxes, you can use the **LISTBOX SET COLUMN FORMULA** command instead.

If this command is applied to a data source that is not modifiable, it does nothing.

## Example

Modification of the data source for an entry area:

```
C_POINTER($ptrField)
$ptrField:=Field(3;2)
OBJECT SET DATA SOURCE(*;"Input";$ptrField)
```

# OBJECT SET DRAG AND DROP OPTIONS

OBJECT SET DRAG AND DROP OPTIONS ( {* ;} object ; draggable ; automaticDrag ; droppable ; automaticDrop )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| draggable | Boolean | ⇒ | Draggable = True; otherwise, False |
| automaticDrag | Boolean | ⇒ | Automatic Drag = True; otherwise, False |
| droppable | Boolean | ⇒ | Droppable = True; otherwise, False |
| automaticDrop | Boolean | ⇒ | Automatic Drop = True; otherwise, False |

## Description

The **OBJECT SET DRAG AND DROP OPTIONS** command sets or dynamically modifies the drag and drop options for the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

In each parameter, you pass a Boolean indicating whether the corresponding option is enabled or disabled:

- *draggable* = True: Object draggable in programmed mode.
- *automaticDrag* = True (only used with text fields and variables, combo boxes and list boxes): Object draggable in automatic mode.
- *droppable* = True: Object accepts drops in programmed mode.
- *automaticDrop* = True (only used with picture fields and variables, text, combo boxes and list boxes): Object accepts drops in automatic mode.

## Example

Setting a text area to automatic drag and drop:

```
OBJECT SET DRAG AND DROP OPTIONS(*;"Comments";False;True;False;True)
```

# OBJECT SET ENABLED

OBJECT SET ENABLED ( {* ;} object ; active )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| active | Boolean | ⇒ | True = object(s) enabled; otherwise, False |

## Description

The **OBJECT SET ENABLED** command can be used to enable or disable the object or group of objects specified by *object* in the current form. An enabled object reacts to mouse clicks and to keyboard shortcuts.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you pass a variable reference (object variable only) instead of a string.

Pass True in the *active* parameter to enable the objects and False to disable them.

This command can be applied to the following types of objects:

- Button, Default button, 3D button, Invisible button, Highlight button
- Radio button, 3D radio button, Picture button
- Check Box, 3D Check Box
- Pop-up menu, Drop-down List, Combo Box, Menu/Drop-down List
- Thermometer, Ruler

**Note:** This command has no effect with an object to which a standard action has been assigned (4D looks after modifying the state of this object when necessary), except in the case of the **Validate** and **Cancel** actions.

# OBJECT SET ENTERABLE

OBJECT SET ENTERABLE ( {* ;} object ; enterable )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Table or Field or Variable (if * is omitted) |
| enterable | Boolean | ⇒ | True for enterable; False for non-enterable |

## Description

The **OBJECT SET ENTERABLE** command makes the form objects specified by *object* either enterable or non-enterable.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a table, field or variable in *object*. In this case, specify a table, field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the **Object Properties** section.

Using this command is equivalent to selecting Enterable for a field or variable in the Form Editor's Property List window. This command works in subforms only if it is in the form method of the subform.

When the *entryArea* is enterable (TRUE), the user can move the cursor into the area and enter data. When the *entryArea* is non-enterable (FALSE), the user cannot move the cursor into the area and cannot enter data.

The **OBJECT SET ENTERABLE** command can also be used to enable the "Enter in List" mode by programming for subforms and list forms displayed using the **MODIFY SELECTION** and **DISPLAY SELECTION** commands:

- For subforms, in the *entryArea* parameter, pass either the name of the subform table or the name of the subform object itself, for example: **OBJECT SET ENTERABLE**(*;"Subform";True).
- For list forms, you must pass the name of the form table in the *entryArea* parameter, for example: **OBJECT SET ENTERABLE**([MyTable];True).

Making an object non-enterable does not prevent you from changing its value programmatically.

**Note:** To make a list box cell non-enterable, you pass the value -1 to $0 in the On Before Data Entry event, see **Managing entry**.

## Example 1

The following example sets a shipping field, depending on the weight of the shipment. If the shipment is 1 ounce or less, then the shipper is set to US Mail and the field is set to be non-enterable. Otherwise, the field is set to be enterable.

```
If([Shipments]Weight<=1)
   [Shipments]Shipper:="US Mail"
   OBJECT SET ENTERABLE([Shipments]Shipper;False)
Else
   OBJECT SET ENTERABLE([Shipments]Shipper;True)
End if
```

## Example 2

Here is the object method of a checkbox located in the header of a list in order to control the Enter in List mode:

```
C_BOOLEAN(bEnterable)
OBJECT SET ENTERABLE([Table1];bEnterable)
```

# OBJECT SET EVENTS

OBJECT SET EVENTS ( {* ;} object ; arrEvents ; mode )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name or "" to designate the form (if * is specified) or<br>Field or variable (if * is omitted) |
| arrEvents | Longint array | ⇒ | Array of events to set |
| mode | Longint | ⇒ | Activation mode for events defined in arrEvents |

## Description

The **OBJECT SET EVENTS** command modifies, for the current process, the configuration of the form events of the form or object(s) designated by the *object* and * parameters.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).
To define the configuration of events for the form itself, pass the optional * parameter and an empty string "" in *object*: in this case, you designate the current form.

**Note:** If you want to modify the events of a subform related to a table, you can only use the syntax based on the object name.

In the *arrEvents* parameter, pass a Longint array containing the list of predefined or custom form events that you want to modify (you can use the *mode* parameter to specify whether the modification consists of enabling or disabling the events). To designate a predefined event to modify, you can pass, in each element of the *arrEvents* array, one of the following constants, found in the "**Form Events**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| On Activate | Longint | 11 | The form’s window becomes the frontmost window |
| On After Edit | Longint | 45 | The contents of the enterable object that has the focus has just been modified |
| On After Keystroke | Longint | 28 | A character is about to be entered in the object that has the focus. **Get edited text** returns the object's text including this character. |
| On After Sort | Longint | 30 | (*List box only*) A standard sort has just been carried out in a list box column |
| On Arrow Click | Longint | 38 | (*3D buttons only*) The “arrow” area of a 3D button is clicked |
| On Before Data Entry | Longint | 41 | (*List box only*) A list box cell is about to change to editing mode |
| On Before Keystroke | Longint | 17 | A character is about to be entered in the object that has the focus. **Get edited text** returns the object's text without this character. |
| On Begin Drag Over | Longint | 46 | An object is being dragged |
| On Begin URL Loading | Longint | 47 | (*Web areas only*) A new URL is loaded in the Web area |
| On bound variable change | Longint | 54 | The variable bound to a subform is modified. |
| On Clicked | Longint | 4 | A click occurred on an object |
| On Close Box | Longint | 22 | The window’s close box has been clicked |
| On Close Detail | Longint | 26 | You left the detail form and are going back to the output form |
| On Collapse | Longint | 44 | (*Hierarchical lists and hierarchical list boxes*) An element of the hierarchical list or hierarchical list box has been collapsed using a click or a keystroke |
| On Column Moved | Longint | 32 | (*List box only*) A list box column is moved by the user via drag and drop |
| On Column Resize | Longint | 33 | (*List box only*) The width of a list box column is modified by a user with the mouse |
| On Data Change | Longint | 20 | Object data has been modified |
| On Deactivate | Longint | 12 | The form’s window ceases to be the frontmost window |
| On Delete Action | Longint | 58 | (*Hierarchical lists and List boxes*) The user attempts to delete an item |
| On Display Detail | Longint | 8 | A record is about to be displayed in a list or a row is about to be displayed in a list box. |
| On Double Clicked | Longint | 13 | A double click occurred on an object |
| On Drag Over | Longint | 21 | Data could be dropped onto an object |
| On Drop | Longint | 16 | Data has been dropped onto an object |
| On End URL Loading | Longint | 49 | (*Web areas only*) All the resources of the URL have been loaded |
| On Expand | Longint | 43 | (*Hierarchical lists and hierarchical list boxes*) An element of the hierarchical list or hierarchical list box has been expanded using a click or a keystroke |
| On Footer Click | Longint | 57 | (*List boxes only*) A click occurs in the footer of a list box or a list box column |
| On Getting Focus | Longint | 15 | A form object is getting the focus |

| Constant | Type | Value | Comment |
|---|---|---|---|
| On Header | Longint | 5 | The form's header area is about to be printed or displayed |
| On Header Click | Longint | 42 | (*List box only*) A click occurs in a column header of the list box |
| On Load Record | Longint | 40 | During entry in list, a record is loaded during modification (the user clicks on a record line and a field changes to editing mode) |
| On Long Click | Longint | 39 | (*3D buttons only*) A 3D button is clicked and the mouse button remains pushed for a certain lapse of time |
| On Losing Focus | Longint | 14 | A form object is losing the focus |
| On Mac toolbar button | Longint | 55 | The user clicks on the tool bar management button under Mac OS. |
| On Menu Selected | Longint | 18 | A menu item has been chosen |
| On Mouse Enter | Longint | 35 | The mouse cursor enters the graphic area of an object |
| On Mouse Leave | Longint | 36 | The mouse cursor leaves the graphic area of an object |
| On Mouse Move | Longint | 37 | The mouse cursor moves at least one pixel OR a modifier key (Shift, Alt, Shift Lock) was pressed. If the event is checked for an object only, it is generated only when the cursor is within the graphic area of the object |
| On Open Detail | Longint | 25 | The detail form associated with the output form or with the listbox is about to be opened |
| On Open External Link | Longint | 52 | (*Web areas only*) An external URL has been opened in the browser |
| On Outside Call | Longint | 10 | The form received a **CALL PROCESS** call |
| On Picture Scroll | Longint | 59 | The user scrolls the contents of a picture field or variable using the mouse or keyboard. |
| On Plug in Area | Longint | 19 | An external object requested its object method to be executed |
| On Printing Break | Longint | 6 | One of the form's break areas is about to be printed |
| On Printing Detail | Longint | 23 | The form's detail area is about to be printed |
| On Printing Footer | Longint | 7 | The form's footer area is about to be printed |
| On Resize | Longint | 29 | The form window is resized |
| On Row Moved | Longint | 34 | (*List box only*) A list box row is moved by the user via drag and drop |
| On Selection Change | Longint | 31 | • *List box*: The current selection of rows or columns is modified<br>• *Records in list*: The current record or the current selection of rows is modified in a list form or subform<br>• *Hierarchical list*: The selection in the list is modified following a click or a keystroke<br>• *Enterable field or variable*: The text selection or the position of the cursor in the area is modified following a click or a keystroke |
| On Timer | Longint | 27 | The number of ticks defined by the **SET TIMER** command has passed |
| On Unload | Longint | 24 | The form is about to be exited and released |
| On URL Filtering | Longint | 51 | (*Web areas only*) A URL was blocked by the Web area |
| On URL Loading Error | Longint | 50 | (*Web areas only*) An error occurred when the URL was loading |

| Constant | Type | Value | Comment |
|---|---|---|---|
| On URL Resource Loading | Longint | 48 | (*Web areas only*) A new resource is loaded in the Web area |
| On Validate | Longint | 3 | The record data entry has been validated |
| On Window Opening Denied | Longint | 53 | (*Web areas only*) A pop-up window has been blocked |

It is important to note that the On Load event is not included in this list: this event cannot be defined because it has already been generated during the execution of the command.

In *arrEvents*, you can also pass any value corresponding to a custom event. In this case, we recommend using negative values (see the **CALL SUBFORM CONTAINER** command).

You use the *mode* parameter to set the overall processing to be carried out for the array elements. To do this, you can pass one of the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Disable events others unchanged | Longint | 2 | All the events listed in the *arrEvents* array are disabled; the status of other events remain unchanged |
| Enable events disable others | Longint | 0 | All the events listed in the *arrEvents* array are enabled; all the other events are disabled |
| Enable events others unchanged | Longint | 1 | All the events listed in the *arrEvents* array are enabled; the status of other events remain unchanged |

The **OBJECT SET EVENTS** command can lead to the enabling of events that are not supported by the *object* (depending on its type). In this case, the events will simply be ignored.

If an *object* is duplicated after a call to the **OBJECT SET EVENTS** command, the resulting enabled/disabled configuration is also duplicated.

## Example 1

Enabling three form events for a set of list box objects and disabling other events:

```
ARRAY LONGINT($MyEventsOnLB;3)
$MyEventsOnLB{1}:=On After Sort
$MyEventsOnLB{2}:=On Column Moved
$MyEventsOnLB{3}:=On Column Resize
OBJECT SET EVENTS(*;"MyLB@";$MyEventsOnLB;Enable events disable others)
 // enables 3 events and disables all others
```

## Example 2

Disabling three form events for a set of list box objects, without modifying the other events:

```
ARRAY LONGINT($MyEventsOnLB;3)
$MyEventsOnLB{1}:=On After Sort
$MyEventsOnLB{2}:=On Column Moved
$MyEventsOnLB{3}:=On Column Resize
OBJECT SET EVENTS(*;"MyLB@";$MyEventsOnLB;Disable events others unchanged)
 // disables only these 3 events
```

## Example 3

Enabling a form event for an object, without modifying the other events:

```
ARRAY LONGINT($MyEventsOnLB;1)
$MyEventsOnLB{1}:=On Column Moved
OBJECT SET EVENTS(*;"Col1";$MyEventsOnLB;Enable events others unchanged)
 // only enables this event
```

## Example 4

Disabling all events of the form:

```
ARRAY LONGINT($MyFormEvents;0)
OBJECT SET EVENTS(*;"";$MyFormEvents;Enable events disable others)
 // disables all events
```

## Example 5

Disables a single event of the form without modifying the others:

```
ARRAY LONGINT($MyFormEvents;1)
$MyFormEvents{1}:=On Timer
OBJECT SET EVENTS(*;"";$MyFormEvents;Disable events others unchanged)
 // only disables this event
```

# OBJECT SET FILTER

OBJECT SET FILTER ( {* ;} object ; entryFilter )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Field or Variable (if * is omitted) |
| entryFilter | String | ⇒ | New data entry filter for the enterable area |

## Description

**OBJECT SET FILTER** sets the entry filter for the objects specified by *object* to the filter you pass in *entryFilter*.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the **Object Properties** section.

**OBJECT SET FILTER** can be used for input and dialog forms and can be applied to fields and enterable variables that accept an entry filter in the Design environment.

Passing an empty string in *entryFilter* removes the current entry filter for the objects.

**Note:** This command cannot be used with fields located in a subform's list form.

**Note:** In *entryFilter*, to use entry filters you may have predefined using the Tool Box, prefix the name of the filter with a vertical bar (|).

## Example 1

The following example sets the entry filter for a postal code field. If the address is in the U.S., the filter is set to ZIP codes. Otherwise, it is set to allow any entry:

```
If([Companies]Country="US") ` Set the filter to a ZIP code format
   OBJECT SET FILTER([Companies]ZIP Code:"&9#####")
Else ` Set the filter to accept alpha and numeric and uppercase the alpha
   OBJECT SET FILTER([Companies]ZIP Code:"~@")
End if
```

## Example 2

The following example allows only the letters "a," "b," "c," or "g" to be entered in two places in the field Field:

```
OBJECT SET FILTER([Table]Field ;"&"+Char(Double quote)+"a;b;c;g"+Char(Double quote)+"##")
```

**Note:** This example sets the entry filter to *&"a;b;c;g"##*.

# OBJECT SET FOCUS RECTANGLE INVISIBLE

OBJECT SET FOCUS RECTANGLE INVISIBLE ( {* ;} object ; invisible )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| invisible | Boolean | ⇒ | True = focus rectangle hidden,<br>False = focus rectangle shown |

## Description

The **OBJECT SET FOCUS RECTANGLE INVISIBLE** command sets or dynamically modifies the visibility option for the focus rectangle of the object(s) designated by the *object* and * parameters for the current process. This setting corresponds to the **Hide focus rectangle** option that is available for enterable objects in the Property List in the Design mode.

**Note:** You can only use this option under Mac OS. It has no effect under Windows.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable or a field. In this case, you pass a variable reference instead of a string.

Pass **True** in the *invisible* parameter to hide the focus rectangle and **False** to keep it visible.

# OBJECT SET FONT

OBJECT SET FONT ( {* ;} object ; font )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Field or Variable (if * is omitted) |
| font | String | ⇒ | Font name |

## Description

**OBJECT SET FONT** displays the *object* using the font specified in the *font* parameter. The *font* parameter must contain a valid font name.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string.

## Example 1

The following example sets the font for a button named *bOK*:

```
OBJECT SET FONT(bOK;"Arial")
```

## Example 2

The following example sets the font for all the form objects whose name contains "info":

```
OBJECT SET FONT(*;"@info@";"Times")
```

## Example 3

The following example uses the special *%password* option, designed for entry and display of “password” type fields. When you pass "%password" in the *font* parameter:

- every character entered in the object is displayed with the same symbol,
- "copy" and "cut" actions are disabled in the object.

**Note:** You can use the *%password* option with field, variable and combo box type objects.

```
OBJECT SET FONT([Users]Password;"%password")
```

# OBJECT SET FONT SIZE

OBJECT SET FONT SIZE ( {* ;} object ; size )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Field or Variable (if * is omitted) |
| size | Longint | ⇒ | Font size in points |

## Description

**OBJECT SET FONT SIZE** sets the form objects specified by *object* to be displayed using the font size you pass in *size*.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the **Object Properties** section.

The *size* is any integer between 1 and 255. If the exact font size does not exist, characters are scaled.

The area for the object, as defined in the form, must be large enough to display the data in the new size. Otherwise, the text may be truncated or not displayed at all.

## Example 1

The following example sets the font size for a variable named *vtInfo*:

```
OBJECT SET FONT SIZE(vtInfo;14)
```

## Example 2

The following example sets the font size for all the form objects whose name starts with "hl":

```
OBJECT SET FONT SIZE(*;"hl@";14)
```

# OBJECT SET FONT STYLE

OBJECT SET FONT STYLE ( {* ;} object ; styles )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Field or Variable (if * is omitted) |
| styles | Longint | ⇒ | Font style |

## Description

**OBJECT SET FONT STYLE** sets the form objects specified by *object* to be displayed using the font style you pass in *styles*.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the **Object Properties** section.

You pass in *styles* a sum of the constants describing your font style selection. The following are the predefined constants provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Bold | Longint | 1 |
| Italic | Longint | 2 |
| Plain | Longint | 0 |
| Underline | Longint | 4 |

## Example 1

This example sets the font style for a button named *bAddNew*. The font style is set to bold italic:

```
OBJECT SET FONT STYLE(bAddNew;Bold+Italic)
```

## Example 2

This example sets the font style to Plain for all form objects with names starting with “vt”:

```
OBJECT SET FONT STYLE(*;"vt@";Plain)
```

# OBJECT SET FORMAT

OBJECT SET FORMAT ( {* ;} object ; displayFormat )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Field or Variable (if * is omitted) |
| displayFormat | String | ⇒ | New display format for the object |

## Description

**OBJECT SET FORMAT** sets the display format for the objects specified by *object* to the format you pass in *displayFormat*. The new format is only used for the current display; it is not stored with the form.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the **Object Properties** section.

**OBJECT SET FORMAT** can be used for both input forms and output forms (displayed or printed) and can be applied to fields or variables (enterable/non-enterable).

Naturally, you must use a display format compatible with the type of data found in the object or with the object itself.

### Boolean

To format Boolean fields, there are two possibilities:

- You can pass a single value in *displayFormat*. In this case, the field will be displayed as a checkbox and its label will be the value specified.
- You can pass two values, separated by a semicolon (*;*), in *displayFormat*. In this case, the field will be displayed as two radio buttons.

### Date

To format Date fields or variables, pass **Char(n)** in *displayFormat*, where *n* is one of the following predefined constants provided by 4D:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Blank if null date | Longint | 100 | "" instead of 0 |
| Date RFC 1123 | Longint | 10 | |
| Internal date abbreviated | Longint | 6 | Dec 29, 2006 |
| Internal date long | Longint | 5 | December 29, 2006 |
| Internal date short | Longint | 7 | 12/29/2006 |
| Internal date short special | Longint | 4 | 12/29/06 (but 12/29/1896 or 12/29/2096) |
| ISO Date | Longint | 8 | 2006-12-29T00:00:00 (deprecated) |
| ISO Date GMT | Longint | 9 | 2010-09-13T16:11:53Z |
| System date abbreviated | Longint | 2 | Sun, Dec 29, 2006 |
| System date long | Longint | 3 | Sunday, December 29, 2006 |
| System date short | Longint | 1 | 12/29/2006 |

**Note:** The Blank if null date constant must be added to the format; it indicates that in the case of a null value, 4D must display an empty area instead of zeros.

### Time

To format Time fields or variables, pass **Char(n)** in *displayFormat*, where *n* is one of the following predefined constants provided by 4D:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Blank if null time | Longint | 100 | "" instead of 0 |
| HH MM | Longint | 2 | 01:02 |
| HH MM AM PM | Longint | 5 | 1:02 AM |
| HH MM SS | Longint | 1 | 01:02:03 |
| Hour min | Longint | 4 | 1 hour 2 minutes |
| Hour min sec | Longint | 3 | 1 hour 2 minutes 3 seconds |
| ISO time | Longint | 8 | 0000-00-00T01:02:03 |
| Min sec | Longint | 7 | 62 minutes 3 seconds |
| MM SS | Longint | 6 | 62:03 |
| System time long | Longint | 11 | 1:02:03 AM HNEC (Mac only) |
| System time long abbreviated | Longint | 10 | 1•02•03 AM (Mac only) |
| System time short | Longint | 9 | 01:02:03 |

**Note:** The Blank if null time constant must be added to the format; it indicates that in the case of a null value, 4D must display an empty area instead of zeros.

### Picture

To format Picture fields or variables, pass **Char(n)** in *displayFormat*, where *n* is one of the following predefined constants provided by 4D:

| Constant | Type | Value |
|---|---|---|
| On background | Longint | 3 |
| Replicated | Longint | 7 |
| Scaled to fit | Longint | 2 |
| Scaled to fit prop centered | Longint | 6 |
| Scaled to fit proportional | Longint | 5 |
| Truncated centered | Longint | 1 |
| Truncated non centered | Longint | 4 |

### Alpha and number

To format fields or variables of the Alpha or Number type, pass the label of the format directly in the *displayFormat* parameter.

For more information about display formats, see the **Number formats** and **Alpha formats** sections of the 4D *Design Reference* manual.

**Note:** In *displayFormat*, to use custom display formats that you may have created in the tool box, prefix the name of the format with a vertical bar (|).

### Picture buttons

To format picture buttons, in the *displayFormat* parameter, pass a character string respecting the following syntax:
*cols;lines;picture;flags{;ticks}*

- *cols* = number of columns in the picture.
- *lines* = number of lines in the picture.
- *picture* = picture used, coming from the picture library or a picture variable:
  - If the picture comes from the picture library, enter its number, preceded by a question mark (e.g.: "?250").
  - If the picture comes from a picture variable, enter the variable name.
- *flags* = display mode and operation of a picture button. This parameter can take any of the following values: 0, 1, 2, 16, 32, 64 and 128. Each of these values represents a display mode or an operation mode. These values are cumulative; for instance, if you want to enable the modes 1 and 64, pass 65 in the *flags* parameter. Here are the details for each value:
  - *flags* = 0 (no option)
    Displays the next picture in the series when the user clicks the picture. Displays the previous picture in the series when the user holds down the Shift key and clicks on the picture. When the user reaches the last picture in the series, the picture does not change when the user clicks it again. That is, it does not cycle back to the first picture in the series.

    - *flags* = 1 (Switch Continuously)
      Similar to the previous option except that the user can hold down the mouse button to display the pictures continuously (i.e., as an animation). When the user reaches the last picture, the object does not cycle back to the first picture.
    - *flags* = 2 (Loop Back to First Frame)
      Similar to the previous option except that the pictures are displayed in a continuous loop. When the user reaches the last picture and clicks again, the first picture appears, and so forth.
    - *flags* = 16 (Switch when Roll Over)
      The contents of the picture button are modified when the mouse cursor passes over it. The initial picture is re-established when the cursor leaves the button's area. This mode is frequently used in multimedia applications or in HTML documents. The picture that is then displayed is the last picture of the thumbnail table, unless the Use Last Frame as Disabled option is selected (128). If that option is selected, it is the next-to-last thumbnail that is displayed.
    - *flags* = 32 (Switch Back when Released)
      This mode operates with two pictures. It displays the first picture all the time except when the user clicks the button. In that case, the second picture is displayed until the mouse button is released, whereupon it switches back to the first picture. This mode allows you to create an action button that displays its status (idle or clicked). You can use this mode to create a 3D effect or display any picture that depicts the action.
    - *flags* = 64 (Transparent)
      Used to make the background picture transparent.
    - *flags* = 128 (Use Last Frame as Disabled)
      This mode allows you to set the last thumbnail as the thumbnail to display when the button is disabled. When this mode is selected, 4D displays the last thumbnail when the button is disabled. When this mode is used in addition to the modes 0, 1 and 2, the last thumbnail is not taken into account in the sequence of the other modes. It will appear only when the button is disabled.
- *ticks* = activates the “Switch every n Ticks” mode and sets the time interval between the display of each picture. When this optional parameter is passed, it allows you to cycle through the contents of the picture button at the specified speed. For example, if you enter "2;3;?16807;0;10", the picture button will display a different picture every 10 ticks. When this mode is active, only the Transparent mode can be used (64).

### Picture pop-up menus

To format picture pop-up menus, in the *displayFormat* parameter, pass a character string respecting the following syntax:
*cols;lines;picture;hMargin;vMargin;flags*

- *cols* = number of columns in the picture.
- *lines* = number of lines in the picture.
- *picture* = picture used, coming from the picture library or a picture variable:
    - If the picture comes from the picture library, enter its number, preceded by a question mark (e.g.: "?250").
    - If the picture comes from a picture variable, enter the variable name.
- *hMargin* = margin in pixels between the horizontal limits of the menu and the picture.
- *vMargin* = margin in pixels between the vertical limits of the menu and the picture.
- *flags* = transparency mode of picture pop-up menu. Accepts the values 0 and 64:
    - *flags* = 0: the picture pop-up menu is not transparent,
    - *flags* = 64: the picture pop-up menu is transparent.

### Thermometers and rulers

To format objects of the thermometer or ruler type, in the *displayFormat* parameter, pass a character string respecting the following syntax:
*min;max;unit;step;flags{;format{;display}}*

- *min* = value of the first graduation of the indicator.
- *max* = value of the last graduation of the indicator.
- *unit* = interval between the indicator graduations.
- *step* = minimum interval of cursor movement in the indicator.
- *flags* = display mode and operation of indicators. This parameter accepts the values 0, 2, 3, 16, 32 and 128. These values can be accumulated in order to set several options (except for 128). Here are the details for each value:
    - *flags* = 0: does not display the units.
    - *flags* = 2: displays the units on the right or below the indicator.
    - *flags* = 3: displays the units on the left or above the indicator.
    - *flags* = 16: displays graduations adjacent to the units.
    - *flags* = 32: On Data Change is executed while the user is adjusting the indicator. If this value is not used, On Data Change occurs only after the user is finished adjusting the indicator.

    - *flags* = 128: activates the "Barber shop" (continuous animation) mode. This value cannot be combined with others. In this mode, the other parameters are ignored (except for the *display* parameter if passed). For more information about this mode, please refer to the *Design Reference* manual.
- *format* = display format of the indicator graduations.
  Keep in mind that the units and graduations are automatically hidden if the size of the indicator object does not permit them to be displayed correctly.
- *display* = specific display options. In the case of thermometers, this parameter is only taken into account when the *flags* subparameter is 128.
    - *display* = 0 (or is omitted): displays a standard ruler / displays a thermometer in continuous animation of the "barber shop" type.
    - *display* = 1 : activates "Stepper" mode for a ruler / activates the "Asynchronous progress" mode for a thermometer. For more information about these options, please refer to the *Design Reference* manual.

### Dials

To format objects of the dial type, in the *displayFormat* parameter, pass a character string respecting the following syntax:
*min;max;unit;step{;flags}*

- *min* = value of the first graduation of the indicator.
- *max* = value of the last graduation of the indicator.
- *unit* = interval between the indicator graduations.
- *step* = minimum interval of cursor movement in the indicator.
- *flags* = operation mode of the dial (optional). This parameter only accepts the value 32: On Data Change is executed while the user is adjusting the indicator. If this value is not used, On Data Change occurs only after the user is finished adjusting the indicator.

### Button grids

To format button grids, in the *displayFormat* parameter, pass a character string respecting the following syntax:
*cols;lines*

- *cols* = number of columns of the grid.
- *lines* = number of lines of the grid.

**Note:** For more information about the display formats for form objects, refer to the 4D Design Reference manual.

### 3D buttons

To format 3D buttons, in the *displayFormat* parameter, pass a character string respecting the following syntax:
*title;picture;background;titlePos;titleVisible;iconVisible;style;horMargin;vertMargin;iconOffset;popupMenu;hyperlink;numStates*

- *title* = Button title. This value can be expressed as text or a resource number (ex.: “:16800,1”)
- *picture* = Picture linked to a button that comes from a picture library or a picture variable:
    - If the picture comes from a picture library, enter its number, preceded with a question mark (ex.: “?250”).
    - If the picture comes from a picture variable, enter the variable name.
    - If the picture comes from a file stored in the Resources folder of the database, enter a URL of the type "#{folder/}picturename" or "file:{folder/}picturename".
- *background* = Background picture linked to a button (Custom style), that comes from a picture library, a picture variable, a PICT resource or a file stored in the Resources folder (see above).
- *titlePos* = position of the button title. Five values are possible:
    - *titlePos* = 1: Left
    - *titlePos* = 2: Top
    - *titlePos* = 3: Right
    - *titlePos* = 4: Bottom
    - *titlePos* = 5: Middle
- *titleVisible* = Defines whether or not the title is visible. Two values are possible:
    - *titleVisible* = 0: the title is hidden
    - *titleVisible* = 1: the title is displayed
- *iconVisible* = Defines whether or not the icon is visible. Two values are possible:
    - *iconVisible* = 0 : the icon is hidden
    - *iconVisible* = 1 : the icon is displayed
- *style* = Button style. The value of this option determines whether various other options are taken into consideration (for example, background). The following values are possible:
    - *style* = 0: None
    - *style* = 1: Background offset

    - *style* = 2: Push button
    - *style* = 3: Toolbar button
    - *style* = 4: Custom
    - *style* = 5: Circle
    - *style* = 6: Small system square
    - *style* = 7: Office XP
    - *style* = 8: Bevel
    - *style* = 9: Rounded bevel
    - *style* = 10: Collapse/Expand
    - *style* = 11: Help
    - *style* = 12: OS X Textured
    - *style* = 13: OS X Gradient
- *horMargin* = Horizontal margin. Number of pixels delimiting the inside left and right margins of the button (areas that the icon and the text must not encroach upon).
- *vertMargin* = Vertical margin. Number of pixels delimiting the inside top and bottom margins of the button (areas that the icon and the text must not encroach upon).
- *iconOffset* = Shifting of the icon to the right and down. This value, expressed in pixels, indicates the shifting of the button icon to the right and down when the button is clicked (the same value is used for both directions).
- *popupMenu* = Association of a pop-up menu with the button. Three values are possible:
    - *popupMenu* = 0: No pop-up menu
    - *popupMenu* = 1: With linked pop-up menu
    - *popupMenu* = 2: With separate pop-up menu
- *hyperlink* = TItle is underlined on mouseover to resemble a hyperlink (legacy mechanism). Two values are possible:
    - *hyperlink* = 0: title is not underlined on mouseover
    - *hyperlink* = 1: title is underlined on mouseover
- *numStates* = Number of states present in picture used as icon for the 3D button, and which will be used by 4D to represent the standard button states (from 0 to 4).

Certain options are not taken into account for all 3D button styles.  Also, in certain cases, you may wish to not change all the options. To not pass an option, simply omit the corresponding value. For example, if you do not want to pass the *titleVisible*, *vertMargin* and *hyperlink* options, you can write:

```
OBJECT SET FORMAT(myVar;"NiceButton:?256::562:1:;1:4:5:;5:0:;2")
```

### List box headers

To format the icon in a list box header, pass a character string in the *displayFormat* parameter, which respects the following syntax:
*picture;iconPos*

- *picture* = header picture, coming from the picture library, a picture variable, or a picture file:
    - If the picture comes from the picture library, enter its number, preceded by a question mark (e.g.: "?250").
    - If it comes from a picture variable, enter the variable name.
    - If it comes from a file stored in the Resources folder of the database, enter a URL of the type "# {folder/}picturename" or "file:{folder/}picturename".
- *iconPos* = position of icon in header. Two values are supported:
    - *iconPos* = 1: Left
    - *iconPos* = 2: Right

This feature is useful, for example, when you want to work with a customized sort icon.

## Example 1

The following line of code formats the *[Employee]Date Hired* field to the fifth format (Internal date long).

```
OBJECT SET FORMAT([Employee]Date Hired;Char(Internal date long))
```

## Example 2

The following example changes the format for a *[Company]ZIP Code* field according to the length of the value stored in the field:

```
If(Length([Company]ZIP Code)=9)
   OBJECT SET FORMAT([Company]ZIP Code;"#####-####")
Else
   OBJECT SET FORMAT([Company]ZIP Code;"#####")
End if
```

## Example 3

The following example formats the value of the [Stats]Results field depending on whether it is a positive, negative, or null number:

```
OBJECT SET FORMAT([Stats]Results;"### ##0.00;(### ##0.00);")
```

## Example 4

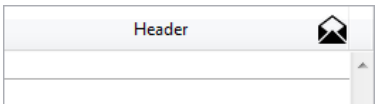
The following example sets the format of a Boolean field to display Married and Unmarried, instead of the default Yes and No:

```
OBJECT SET FORMAT([Employee]Marital Status;"Married;Unmarried")
```

## Example 5

Provided that you have stored a picture file named "envelope_open.png" in the Resources folder of the database, you can write:

```
vIcon:="#envelope_open.png"
vPos:="2" // Right
OBJECT SET FORMAT(*;"Header1";vIcon+";"+vPos)
```




## Example 6

The following example sets the format of a Boolean field to display a checkbox labelled “Classified”:

```
OBJECT SET FORMAT([Folder]Classification;"Classified")
```

## Example 7

You have a table of thumbnails containing 1 row and 4 columns, intended to display a picture button (“default”, “clicked”, “roll over” and “disabled”). You want to associate the Switch when Roll Over, Switch back when Released and Use Last Frame as Disabled options with it:

```
OBJECT SET FORMAT(*;"PictureButton";"4;1;?15000;176")
```

## Example 8

Switching a thermometer to "Barber shop" mode:

```
OBJECT SET FORMAT($Mythermo;";;;;128")
$Mythermo:=1 `Start animation
```

# OBJECT SET HELP TIP

OBJECT SET HELP TIP ( {* ;} object ; helpTip )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| helpTip | Text | ⇒ | Contents of help message |

## Description

The **OBJECT SET HELP TIP** command sets or dynamically modifies the help tip associated with the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

When the form is executed, help messages appear as help tips when the cursor moves over the field or object. Help tips can also be set using the Form editor and the Structure editor in Design mode.

Pass the contents of the message in the *helpTip* parameter. You can pass either:

- a character string, for example "Use a / as separator",
- an empty string "" to remove the help tip.

# OBJECT SET HORIZONTAL ALIGNMENT

OBJECT SET HORIZONTAL ALIGNMENT ( {* ;} object ; alignment )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an Object name (String) If omitted, object is a field or a variable |
| object | Form object | ⇒ | Object name (if * specified), or Field or variable (if * omitted) |
| alignment | Longint | ⇒ | Alignment code |

## Description

The **OBJECT SET HORIZONTAL ALIGNMENT** command allows you to set the type of horizontal alignment applied to the object(s) designated by the *object* and * parameters.

If you specify the optional * parameter, you indicate an object name (a string) in the *object* parameter. If you omit the * parameter, you indicate a field or variable in the *object* parameter. In this case, you specify a field or variable reference (field or variable objects only) instead of a string.

Pass one of the constants of the **Form Objects (Properties)** theme in the *alignment* parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Align center | Longint | 3 | |
| Align default | Longint | 1 | |
| Align left | Longint | 2 | |
| Align right | Longint | 4 | |
| wk justify | Longint | 5 | Available for 4D Write Pro areas only |

The form objects to which alignment can be applied are as follows:

- Scrollable areas
- Combo boxes
- Static text
- Group areas
- Pop up menu/Drop-down lists
- Fields
- Variables
- List boxes
- List box columns
- List box headers
- List box footers
- **4D Write Pro Reference** areas

# OBJECT SET INDICATOR TYPE

OBJECT SET INDICATOR TYPE ( {* ;} object ; indicator )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| indicator | Longint | ⇒ | Indicator type |

## Description

The **OBJECT SET INDICATOR TYPE** command modifies the type of progress indicator for the thermometer(s) designated by the *object* and * parameters in the current process.

The indicator type defines the display variant of the thermometer. For more information, refer to **Indicators** in the *Design Reference* manual.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In the *indicator* parameter, pass the type of indicator you want to display. You can use one of the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Asynchronous progress bar | Longint | 3 | Circular indicator displaying continuous animation |
| Barber shop | Longint | 2 | Bar displaying continuous animation |
| Progress bar | Longint | 1 | Standard progress bar |

# OBJECT SET KEYBOARD LAYOUT

OBJECT SET KEYBOARD LAYOUT ( {* ;} object ; languageCode )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable or field |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable or field (if * is omitted) |
| languageCode | String | ⇒ | RFC3066 ISO639 and ISO3166 language code,<br>"" = no change |

## Description

The **OBJECT SET KEYBOARD LAYOUT** command sets or dynamically modifies the keyboard layout associated with the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable or field. In this case, you pass a reference instead of a name.

In *languageCode*, you pass a string indicating the code of the language to use, based on RFC3066, ISO639 and ISO3166. For more information, refer to the description of the **SET DATABASE LOCALIZATION** command.

# OBJECT SET LIST BY NAME

OBJECT SET LIST BY NAME ( {* ;} object {; listType}; list )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an Object Name (String) If omitted, object is a Field or a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Field or Variable (if * is omitted) |
| listType | Longint | ⇒ | Type of list: Choice list, Required list or Excluded list |
| list | String | ⇒ | Name of the list to use as Choice list or "" to disassociate the list |

## Description

The **OBJECT SET LIST BY NAME** command sets, replaces or disassociates the *list* associated with the object or group of objects specified by *object*. The list whose name is passed in the *list* parameter must have been created using the List Editor in the Design environment.

This command can be applied in an input or dialog form, to fields and enterable variables whose value can be entered as text.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string.

**Note:** This command cannot be used with fields located in a subform's list form.

The **OBJECT SET LIST BY NAME** command now allows you to set or replace all the types of lists associated with the object(s) designated by the *object* and * parameters: choice lists, list of required values, and lists of excluded values. To do this, in the *listType* parameter you pass one of the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Choice list | Longint | 0 | Simple list of values to choose from ("Choice List" option in the Property List) (default) |
| Excluded list | Longint | 2 | Lists values not accepted for entry ("Excluded List" option in the Property List) |
| Required list | Longint | 1 | Lists only values accepted for entry ("Required List" option in the Property List) |

If you omit this parameter, value 0 (Choice list) is used by default.

In the current process, to disassociate a list that was associated with the *object*, pass an empty string ("") in the *list* parameter for the type of list concerned.

## Example 1

The following example sets a choice list for a shipping field. If the shipping is overnight, then the choice list is set to shippers who can ship overnight. Otherwise, it is set to the standard shippers:

```
If([Shipments]Overnight)
   OBJECT SET LIST BY NAME([Shipments]Shipper;"Fast Shippers")
Else
   OBJECT SET LIST BY NAME([Shipments]Shipper;"Normal Shippers")
End if
```

## Example 2

Associate the "color_choice" list as a simple pop-up/drop-down list named "DoorColor":

```
OBJECT SET LIST BY NAME(*;"DoorColor";Choice list;"color_choice")
 // in this case, the 3rd parameter (constant) can be omitted
```

## Example 3

You want to associate the "color_choice" list with the "WallColor" combo box. Since this combo box is enterable, you want for it not to be possible to use certain colors such as "black", "purple" etc. These colors are placed in the "excl_colors" list:

```
OBJECT SET LIST BY NAME(*;"WallColor";Choice list;"color_choice")
OBJECT SET LIST BY NAME(*;"WallColor";Excluded list;"excl_colors")
```

## Example 4

You want to remove the list associations:

```
 // removal of a choice list
OBJECT SET LIST BY NAME(*;"DoorColor";Choice list;"")
 // removal of list of values that are not allowed
OBJECT SET LIST BY NAME(*;"WallColor";Excluded list;"")
```

# OBJECT SET LIST BY REFERENCE

OBJECT SET LIST BY REFERENCE ( {* ;} object {; listType}; list )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| listType | Longint | ⇒ | Type of list: Choice list, Required list or Excluded list |
| list | ListRef | ⇒ | List reference number |

## Description

The **OBJECT SET LIST BY REFERENCE** command defines or replaces the list associated with the object(s) designated by the *object* and * parameters, with the hierarchical list referenced in the *list* parameter.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

By default, if you omit the *listType* parameter, the command defines a source choice list (choice of values) for the object. You can designate any type of list in the *listType* parameter. To do this, you just need to pass one of the following constants found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Choice list | Longint | 0 | Simple list of values to choose from ("Choice List" option in the Property List) (default) |
| Excluded list | Longint | 2 | Lists values not accepted for entry ("Excluded List" option in the Property List) |
| Required list | Longint | 1 | Lists only values accepted for entry ("Required List" option in the Property List) |

In *list*, pass the reference number of the hierarchical list that you want to associated with the object. This list must have been generated using the **Copy list**, **Load list** or **New list** command.

To end the association of a *list* with an *object*, you can just pass 0 in the *list* parameter for the type of list concerned.

Removing a list association does not delete the list reference from memory. Remember to call the **CLEAR LIST** command when you no longer need the list.

This command is particularly interesting in the context of a pop-up or combo box associated with a variable or a field (see the *Design Reference* manual). In this case, the association is dynamic and any change in the list is copied to the form. When the object is associated with an array, the list is copied into the array and any changes to the list are not available automatically (see example 5).

## Example 1

Associating a simple choice list (default list type) to a text field:

```
vCountriesList:=New list
APPEND TO LIST(vCountriesList;"Spain";1)
APPEND TO LIST(vCountriesList;"Portugal";2)
APPEND TO LIST(vCountriesList;"Greece";3)
OBJECT SET LIST BY REFERENCE([Contact]Country;vCountriesList)
```

## Example 2

Associating the "vColor" list as a simple choice list with the "DoorColor" pop-up/drop-down list:

```
vColor:=New list
APPEND TO LIST(vColor;"Blue";1)
APPEND TO LIST(vColor;"Green";2)
APPEND TO LIST(vColor;"Red";3)
APPEND TO LIST(vColor;"Yellow";4)
OBJECT SET LIST BY REFERENCE(*;"DoorColor";Choice list;vColor)
```

## Example 3

Now you want to associate the "vColor" list with a combo box named "WallColor". Since this combo box is enterable, you want to make sure certain colors, such as "black," "purple," etc., cannot be used. These colors are placed in the "vReject" list:

```
OBJECT SET LIST BY REFERENCE(*;"WallColor";Choice list;vColor)
vReject:=New list
APPEND TO LIST(vReject;"Black";1)
APPEND TO LIST(vReject;"Gray";2)
APPEND TO LIST(vReject;"Purple";3)
OBJECT SET LIST BY REFERENCE(*;"WallColor";Excluded list;vReject)
```

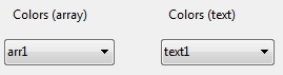
## Example 4

You want to remove the list associations:

```
OBJECT SET LIST BY REFERENCE(*;"WallColor";Choice list;0)
OBJECT SET LIST BY REFERENCE(*;"WallColor";Required list;0)
OBJECT SET LIST BY REFERENCE(*;"WallColor";Excluded list;0)
```

## Example 5

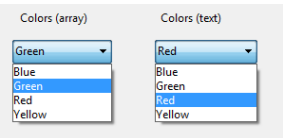
This example illustrates the difference in how the command works when applied to a pop-up menu associated with a text array or one associated with a text variable. There are two pop-up menus in a form:




The contents of these pop-up menus is set using the *<>vColor* list (containing color values). The following code is executed when the form is loaded:

```
ARRAY TEXT(arr1;0) //arr1 pop up
C_TEXT(text1) //text1 pop up
OBJECT SET LIST BY REFERENCE(*;"arr1";<>vColor)
OBJECT SET LIST BY REFERENCE(*;"text1";<>vColor)
```

During execution, both menus propose the same values:

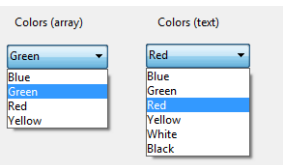



(*Montage showing contents of menus simultaneously*)

Then you run the following code, for example by means of a button:

```
APPEND TO LIST(<>vColor;"White";5)
APPEND TO LIST(<>vColor;"Black";6)
```

Only the menu associated with the Text field is updated (by means of the dynamic reference):




In order to update the list associated with the pop-up managed by array, you need to call the **OBJECT SET LIST BY REFERENCE** command again to copy the contents of the list.

# OBJECT SET MAXIMUM VALUE

OBJECT SET MAXIMUM VALUE ( {* ;} object ; maxValue )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| maxValue | Date, Time, Number | ⇒ | Maximum value for object |

## Description

The **OBJECT SET MAXIMUM VALUE** command modifies the maximum value of the object(s) designated by the *object* and * parameters for the current process.

The "Maximum Value" property can be applied to number, date or time type data. For more information, refer to **Maximum and minimum values** in the *Design Reference* manual.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In *maxValue*, pass the new maximum value you want to assign to the object for the current process. This value must correspond to the object type, otherwise error 18 "Field types are incompatible" is returned.

# OBJECT SET MINIMUM VALUE

OBJECT SET MINIMUM VALUE ( {* ;} object ; minValue )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| minValue | Date, Time, Number | ⇒ | Minimum value for object |

## Description

The **OBJECT SET MINIMUM VALUE** command modifies the minimum value of the object(s) designated by the *object* and * parameters for the current process.

The "Minimum Value" property can be applied to number, date or time type data. For more information, refer to **Maximum and minimum values** in the *Design Reference* manual.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In *minValue*, pass the new minimum value you want to assign to the object for the current process. This value must correspond to the object type, otherwise error 18 "Field types are incompatible" is returned.

# OBJECT SET MULTILINE

OBJECT SET MULTILINE ( {* ;} object ; multiline )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| multiline | Longint | ⇒ | Status of multiline property |

## Description

The **OBJECT SET MULTILINE** command modifies the "Multiline" property of the object(s) designated by the *object* and * parameters.

The "Multiline" property controls two parameters related to the display and printing of text areas: display of words located at the end of the line in single-line areas and the automatic insertion of line returns. For more information, refer to **Multiline** in the *Design Reference* manual. If you apply this command to an object that does not support this property, the command does nothing.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In the *multiline* parameter, pass the new value of the option that you want to set. You can use the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Multiline Auto | Longint | 0 | In single-line areas, words located at the end of lines are truncated and there are no line returns.<br>In multiline areas, 4D carries out automatic line returns. |
| Multiline No | Longint | 2 | There are never line returns: the text is always displayed on a single row. If the Alpha or Text field or variable contains carriage returns, the text located after the first carriage return is removed as soon as the area is modified. |
| Multiline Yes | Longint | 1 | In single-line areas, the text is displayed up to the first carriage return or until the last word that can be displayed entirely. 4D inserts line returns; it is possible to scroll the contents of the area by pressing the down arrow key.<br>In multiline areas, 4D carries out automatic line returns. |

## Example

You want to prohibit multiple lines in an entry area:

```
OBJECT SET MULTILINE(*;"vEntry";Multiline No)
```

# OBJECT SET PLACEHOLDER

OBJECT SET PLACEHOLDER ( {* ;} object ; placeholderText )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| placeholderText | Text | ⇒ | Placeholder text associated with object |

## Description

The **OBJECT SET PLACEHOLDER** command associates placeholder text with the object(s) designated by the *object* and * parameters.

For more information about placeholder text, refer to the *Design Reference* manual.

If placeholder text is already associated with the object through the Property List, this text is replaced in the current process by the contents of the *placeholderText* parameter.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In *placeholderText*, pass the help text or indication that must appear when the object is empty.

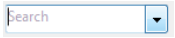
**Note:** The **OBJECT SET PLACEHOLDER** command does not support the insertion of xliff references into the placeholder text. This is only possible for placeholder text that is defined using the Property List.

This command can only be used with form objects of the variable, field or combo box type. You can associate placeholder text with Alpha and Text type values. You can also associate it with Date or Time type data if the form object is given the "Blank if null" property.

## Example

You want to display "Search" as placeholder text in a combo box:

```
OBJECT SET PLACEHOLDER(*;"search_combo";"Search")
```



# OBJECT SET PRINT VARIABLE FRAME

OBJECT SET PRINT VARIABLE FRAME ( {* ;} object ; variableFrame {; fixedSubform} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ➡ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| variableFrame | Boolean | ➡ | True = Variable frame printing, False = Fixed frame printing |
| fixedSubform | Longint | ➡ | Options for printing subforms in fixed size |

## Description

The **OBJECT SET PRINT VARIABLE FRAME** command modifies the Print Variable Frame property of the object(s) designated by the *object* and * parameters.

This property is available for the following objects:

- Text or Picture type variables and fields (see **Print Variable Frame** in the *Design Reference* manual)
- 4D Write Pro areas (see **Using a 4D Write Pro area** in the 4D Write Pro reference manual).
- Subforms. Subforms have an additional option for fixed size printing (see **Subform Printing** in the *Design Reference* manual); the command can be used to configure this option using the *fixedSubform* parameter.

If you apply this command to an object that does not support this property, the command does nothing.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

Pass a Boolean in the *variableFrame* parameter: if you pass **True**, the object is printed with a variable frame. If you pass **False**, it is printed with a fixed frame.

The optional *fixedSubform* parameter lets you set an additional option when you pass **False** in the *variableFrame* parameter and the *object* is a subform (it is ignored in all other cases). In this case, you can define the fixed frame printing mode for the subform. You can pass one of the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Print Frame fixed with multiple records | Longint | 2 | The frame remains the same size, but 4D prints the form several times to include all the records. |
| Print Frame fixed with truncation | Longint | 1 | 4D prints only the records that fit into the area of the subform. The form is printed only once and those records that are not printed are ignored. |

# OBJECT SET RESIZING OPTIONS

OBJECT SET RESIZING OPTIONS ( {* ;} object ; horizontal ; vertical )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| horizontal | Longint | ⇒ | Horizontal resizing option |
| vertical | Longint | ⇒ | Vertical resizing option |

## Description

The **OBJECT SET RESIZING OPTIONS** command sets or dynamically modifies the resizing options for the object(s) designated by the *object* and * parameters for the current process. These options specify how the object is displayed when the form window is resized.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

In the *horizontal* parameter, you pass a value indicating the horizontal resizing option that you want to specify for the *object*. You can pass one of the following constants, found in the **Form Objects (Properties)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Resize horizontal grow | Longint | 1 | If the window grows by 50% in width, the object is expanded by 50% to the right. |
| Resize horizontal move | Longint | 2 | If the window grows by 100 pixels in width, the object is moved 100 pixels to the right. |
| Resize horizontal none | Longint | 0 | If the window is expanded in width, neither the width nor the position of the object changes. |

In the *vertical* parameter, you pass a value indicating the vertical resizing option that you want to specify for the *object*. You can pass one of the following constants, found in the **Form Objects (Properties)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Resize vertical grow | Longint | 1 | If the window grows by 50% in height, the object is lengthened by 50% towards the bottom. |
| Resize vertical move | Longint | 2 | If the window grows by 100 pixels in height, the object is moved 100 pixels towards the bottom. |
| Resize vertical none | Longint | 0 | If the window is expanded in height, neither the height nor the position of the object changes. |

# OBJECT SET RGB COLORS

OBJECT SET RGB COLORS ( {* ;} object ; foregroundColor ; backgroundColor {; altBackgrndColor} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Field or Variable (if * is omitted) |
| foregroundColor | Longint | ⇒ | RGB color value for Foreground color |
| backgroundColor | Longint | ⇒ | RGB color value for Background color |
| altBackgrndColor | Longint | ⇒ | RGB color value for Alternating background color |

## Description

The **OBJECT SET RGB COLORS** command changes the foreground and background colors of the objects specified by the *object* parameter and the optional * parameter. When the command is applied to a List box object, an additional parameter lets you modify the alternating color of the rows.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the **Object Properties** section.

The optional *altBackgrndColor* parameter lets you set an alternate background color for even-numbered rows. This parameter is only used when the object specified is a List box or a column of the List box. When this parameter is used, the *backgroundColor* parameter is only used for odd-numbered rows. Using alternating colors makes lists easier to read.

If *object* specifies a List box object, alternating colors are used for the entire List box. If *object* specifies a column of the List box, only the column will use the colors set.

You indicate RGB color values in *foregroundColor*, *backgroundColor* and *altBackgrndColor*. An RGB value is a 4-byte Long Integer whose format (*0x00RRGGBB*) is described in the following table (bytes are numbered from 0 to 3, from right to left):

| Byte | Description |
|---|---|
| 3 | Must be zero if absolute RGB color |
| 2 | Red component of the color (0..255) |
| 1 | Green component of the color (0..255) |
| 0 | Blue component of the color (0..255) |

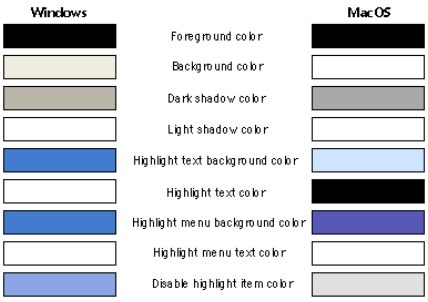
The following table shows some examples of RGB color values:

| Value | Description |
|---|---|
| *0x00000000* | Black |
| *0x00FF0000* | Bright Red |
| *0x0000FF00* | Bright Green |
| *0x000000FF* | Bright Blue |
| *0x007F7F7F* | Gray |
| *0x00FFFF00* | Bright Yellow |
| *0x00FF7F7F* | Red Pastel |
| *0x00FFFFFF* | White |

Alternatively, you can specify one of the “system” colors used by 4D for drawing objects whose colors are set automatically. The following predefined constants are provided by 4D:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Background color | Longint | -2 | This constant can only be used with the *backgroundColor* and *altBackgrndColor* parameters. |
| Background color none | Longint | -16 | |
| Dark shadow color | Longint | -3 | |
| Disable highlight item color | Longint | -11 | |
| Foreground color | Longint | -1 | |
| Highlight menu background color | Longint | -9 | |
| Highlight menu text color | Longint | -10 | |
| Highlight text background color | Longint | -7 | |
| Highlight text color | Longint | -8 | |
| Light shadow color | Longint | -4 | |

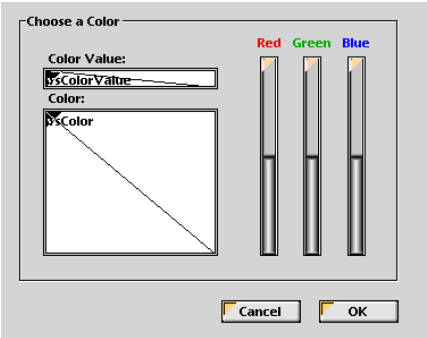
For example, you can get the following colors for enterable field or variable type objects on a standard system:




**WARNING:** These automatic colors depend on the system as well as the type of object to which they are assigned. Depending on the OS version or whether you customize your system colors, 4D will adjust its automatic colors accordingly. Use the automatic color values for setting objects to the system colors, not for setting them to the example colors shown above.

## Example 1

This form contains the two non-enterable variables *vsColorValue* and *vsColor* as well as the three thermometers: *thRed*, *thGreen*, and *thBlue*.




Here are the methods for these objects:

```
  ` vsColorValue non-enterable Object Method
Case of
   :(Form event=On Load)
      vsColorValue:="0x00000000"
End case
  ` vsColor non-enterable variable Object Method
Case of
   :(Form event=On Load)
      vsColor:=""
      OBJECT SET RGB COLORS(vsColor;0x00FFFFFF;0x0000)
End case

  ` thRed Thermometer Object Method
```

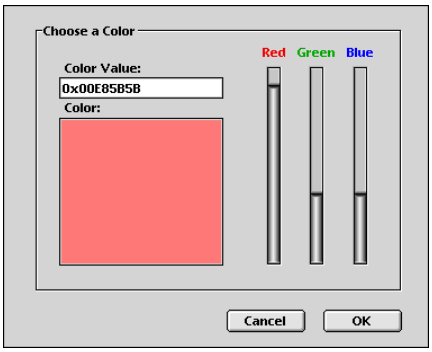
```
CLICK IN COLOR THERMOMETER

` thGreen Thermometer Object Method
CLICK IN COLOR THERMOMETER

` thBlue Thermometer Object Method
CLICK IN COLOR THERMOMETER
```

The project method called by the three thermometers is:

```
` CLICK IN COLOR THERMOMETER Project Method
OBJECT SET RGB COLORS(vsColor;0x00FFFFFF;(thRed<<16)+(thGreen<<8)+thBlue)
vsColorValue:=String((thRed<<16)+(thGreen<<8)+thBlue;"&x")
If(thRed=0)
   vsColorValue:=Substring(vsColorValue;1;2)+"0000"+Substring(vsColorValue;3)
End if
```

Note the use of the **Bitwise Operators** for calculating the color value from the thermometer values.

When executed, the form looks like this:




## Example 2

Changing to transparent background with a light font color:


```
OBJECT SET RGB COLORS(*;"myVar";Light shadow color;Background color none)
```

# OBJECT SET SCROLL POSITION

OBJECT SET SCROLL POSITION ( * ; object {; vPosition {; hPosition}}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a table, a field or a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Table or field or variable (if * is omitted) |
| vPosition | Longint | ⇒ | Line number to display or Vertical scrolling in pixels (pictures) |
| hPosition | Longint | ⇒ | Column number to display (list box) or Horizontal scrolling in pixels (pictures) |
| * | Operator | ⇒ | Display of line (and column if the hPosition parameter is passed) in first position after scroll |

## Description

The **OBJECT SET SCROLL POSITION** command allows scrolling the contents of several types of objects: the lines of a subform, of a list form (displayed using the **MODIFY SELECTION** or **DISPLAY SELECTION** commands), or of a hierarchical list, the rows and columns of a list box or even the pixels of a picture.

**Note:** Scrolling via programming remains possible even if scrollbars have been hidden in the form.

If you pass the first optional * parameter, you indicate that the *object* parameter is the name of a subform, a hierarchical list, a list box object or a picture field/variable (in this case, pass a string in *object*). If you do not pass anything in this parameter, you indicate that the *object* parameter is a table (list form table or subform table), a variable (*ListRef* of a hierarchical list or list box or picture) or a field.

The *vPosition* parameter can be used to specify the number of the row to display or, in the case of a picture, the vertical coordinate of the pixel to display.
If you do not pass this parameter, the command provokes the vertical scroll of lines of the list so that the first highlighted line in the list is visible. In this case, if no line is selected or if at least one selected line is already visible, no vertical scrolling is applied.
If you pass this parameter, the command provokes the vertical scroll of lines of the list so that the set line is visible (highlighted or not). If the line is already visible, the command does nothing, unless the second * parameter is passed (see below).

- For list forms and subforms, this number is the number of the line among the current selection (its position).
- In the case of hierarchical lists, the command takes the expanded/collapsed state of the items into account.
- For list boxes, this number is the number of the row among all the object rows (including hidden rows). If the number passed in *vPosition* corresponds to a hidden row in the list box, the command displays the first visible row that follows.

  **Note:** Keep in mind that this command goes by the "standard" representation (non-hierarchical) of a list box, even if it is displayed in hierarchical mode. Therefore, the result may be different depending on whether the list box is displayed in standard or hierarchical mode (see example).
- For pictures displayed in the form, *vPosition* indicates the vertical coordinate point of the picture to display in the object. Pass 0 in *vPosition* if you do not want to scroll the picture vertically. The value must be expressed in pixels in relation to the origin of the picture. If the vertical coordinate point is already shown in the object, the command does nothing (except when you pass the second * parameter. see below). The picture must be displayed in the "Truncated (non-centered)" format.

The *hPosition* parameter can be used in the context of a list box or a picture.

- For list boxes, you can pass a column number in *hPosition*. Executing the command causes horizontal scrolling of the list box so that this column will be shown. If the column is already visible, the command does nothing. As with vertical scrolling, if you pass the second optional * parameter, the column made visible by the command (if the list box is actually scrolled) will be placed in the first position (see below).
- For a picture displayed in a form, *hPosition* indicates the horizontal coordinate point to display in the object. The value must be expressed in pixels in relation to the origin of the picture. If the horizontal coordinate point is already shown in the object, the command does nothing (except when you pass the second * parameter. see below).

If you pass the second optional * parameter:

- the line made visible using the command (if the list was scrolled) will be placed in the first position of the list. If the line is situated at the end of the list, this option has no effect.

- in the context of a picture, the coordinates requested will be placed at the origin of the picture variable (0,0), even if these coordinates were already shown in the object.

**Note:** The **HIGHLIGHT RECORDS** command features an optional * parameter that allows delegating scroll management to the **OBJECT SET SCROLL POSITION** command.

## Example 1

This example illustrates the difference in the way the command functions depending on whether the list box is displayed in standard or hierarchical mode:

```
OBJECT SET SCROLL POSITION(*;"mylistbox";4;2;*) // displays 4th row of 2nd column of list box in the first position
```

If this statement is applied to a list box displayed in standard mode:

| | | | | |
|---|---|---|---|---|
| France | Brittany | Brest | 120000 | ... |
| France | Brittany | Quimper | 80000 | ... |
| France | Brittany | Rennes | 200000 | ... |
| France | **Normandy** | Caen | 220000 | ... |
| France | Normandy | Deauville | 4000 | ... |
| France | Normandy | Cherbourg | 41000 | ... |
| ... | ... | ... | ... | ... |

... the rows and columns of the list box actually scroll:

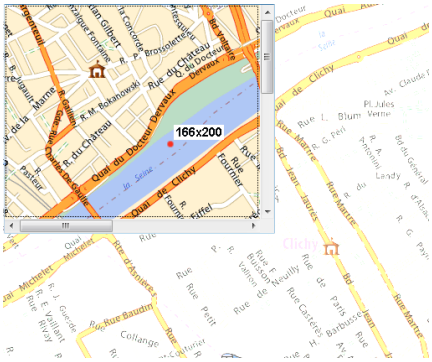
| | | | | |
|---|---|---|---|---|
| Normandy | Caen | 220000 | ... | ... |
| Normandy | Deauville | 4000 | ... | ... |
| **Normandy** | Cherbourg | 41000 | ... | ... |
| ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... |

On the other hand, if the same statement is applied to a list box displayed in hierarchical mode, the rows scroll but not the columns because the 2nd column is part of the hierarchy:

| V **Normandy** | |
|---|---|
| Caen | 220000 |
| Deauville | 4000 |
| Cherbourg | 41000 |
| ... | |

## Example 2

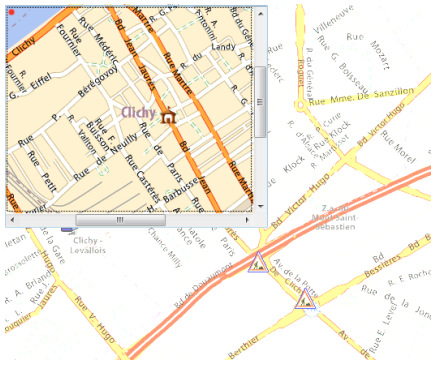
You want to scoll a picture that is included in a form variable. This montage shows the visible part of the picture as well as the point to be displayed (166 pixels vertically and 200 pixels horizontally):




To scroll the visible part and display the red point at the origin of the picture variable, you can just write:

```
OBJECT SET SCROLL POSITION(*;"myVar";166;200;*)
```

You then get the following result:




Make sure that you do not omit the second * parameter in this case, otherwise the picture will not scroll because the point defined is already displayed.

# OBJECT SET SCROLLBAR

OBJECT SET SCROLLBAR ( {* ;} object ; horizontal ; vertical )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| horizontal | Boolean, Longint | ⇒ | Visibility of horizontal scrollbar |
| vertical | Boolean, Longint | ⇒ | Visibility of vertical scrollbar |

## Description

The **OBJECT SET SCROLLBAR** command allows you to display or hide the horizontal and/or vertical scrollbars in the object set using the *object* and * parameters.

If you pass the optional * parameter, you indicate that the *object* parameter is an object name (string). If you do not pass this parameter, you indicate that the *object* parameter is a variable. In this case, you do not pass a string, but a variable reference. For more information about object names, refer to the **Object Properties** section.

In the *horizontal* and *vertical* parameters, you pass values indicating whether the corresponding scrollbars should be displayed. You can pass either Boolean values (True=displayed, False=hidden), or numeric values (0=hidden, 1=displayed, 2=automatic mode). Using numeric values gives you access to the automatic mode, where scrollbars are only displayed when necessary.

The following table indicates the values you can pass in the *horizontal* and *vertical* parameters for objects that accept scrollbars (automatic mode is not available for all objects):

| Objects with scrollbars | Hide scrollbar | Show scrollbar | Automatic mode |
|---|---|---|---|
| Text object fields and variables | False or 0 | True or 1 | *not available* |
| Picture object fields and variables | False or 0 | True or 1 | 2 |
| List boxes | False or 0 | True or 1 | 2 |
| Hierarchical lists | False or 0 | True or 1 | 2 |
| Subforms | False or 0 | True or 1 | *not available* |

By default, scrollbars are displayed.

**Note:** For more information about the automatic mode, refer to **Scroll bars**.

# OBJECT SET SHORTCUT

OBJECT SET SHORTCUT ( {* ;} object ; key {; modifiers} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable or field |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable or field (if * is omitted) |
| key | String | ⇒ | Key to associate with object |
| modifiers | Longint | ⇒ | Modifier key mask or combination of masks |

## Description

The **OBJECT SET SHORTCUT** command sets or dynamically modifies the keyboard shortcut associated with the object(s) designated by the *object* and * parameters for the current process.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable or a field. In this case, you pass a reference instead of a name.

In the *key* parameter, pass a string indicating the key to associate with the object. You can either pass:

- a standard key name, for example "B"
- or a constant (or its value) from the **Shortcut and Associated Keys** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Shortcut with Backspace | String | [backspace] | |
| Shortcut with Carriage Return | String | [return] | |
| Shortcut with Delete | String | [del] | |
| Shortcut with Down arrow | String | [down arrow] | |
| Shortcut with End | String | [end] | |
| Shortcut with Enter | String | [enter] | |
| Shortcut with Escape | String | [esc] | |
| Shortcut with F1 | String | [F1] | |
| Shortcut with F10 | String | [F10] | |
| Shortcut with F11 | String | [F11] | |
| Shortcut with F12 | String | [F12] | |
| Shortcut with F13 | String | [F13] | |
| Shortcut with F14 | String | [F14] | |
| Shortcut with F15 | String | [F15] | |
| Shortcut with F2 | String | [F2] | |
| Shortcut with F3 | String | [F3] | |
| Shortcut with F4 | String | [F4] | |
| Shortcut with F5 | String | [F5] | |
| Shortcut with F6 | String | [F6] | |
| Shortcut with F7 | String | [F7] | |
| Shortcut with F8 | String | [F8] | |
| Shortcut with F9 | String | [F9] | |
| Shortcut with Help | String | [help] | |
| Shortcut with Home | String | [home] | |
| Shortcut with Left arrow | String | [left arrow] | |
| Shortcut with Page down | String | [page down] | |
| Shortcut with Page up | String | [page up] | |
| Shortcut with Right arrow | String | [right arrow] | |
| Shortcut with Tabulation | String | [tab] | |
| Shortcut with Up arrow | String | [up arrow] | |

In the *modifiers* parameter, you can pass one or more modifier keys to associate with the shortcut. To set the *modifiers* parameter, pass one or more of the following "Mask" type constants found in the **Events (Modifiers)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Command key mask | Longint | 256 | Ctrl key under Windows, Command key under OS X |
| Control key mask | Longint | 4096 | Ctrl key under OS X, or right click under Windows and OS X |
| Option key mask | Longint | 2048 | Alt key (also called Option under OS X) |
| Shift key mask | Longint | 512 | Windows and OS X |

**Note:** When you omit the *modifiers* parameter, the object is enabled as soon as you press the key that was set. For example, if you associate the "H" key with a button, this button is enabled whenever you press the H key. This kind of functioning is to be reserved for specific interfaces.

## Example

You want to associate a different shortcut depending on the current language of the application. In the On Load form event, you can write:

```
Case of
      vLang="FR"
      OBJECT SET SHORTCUT(*;"SortButton";"T";Command key mask+Shift key mask) // Ctrl+Shift+T in French
      vLang="US"
      OBJECT SET SHORTCUT(*;"SortButton";"O";Command key mask+Shift key mask) // Ctrl+Shift+O in English
End case
```

# OBJECT SET STYLE SHEET

OBJECT SET STYLE SHEET ( {* ;} object ; styleSheetName )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| styleSheetName | Text | ⇒ | Name of style sheet |

## Description

The **OBJECT SET STYLE SHEET** command modifies, for the current process, the style sheet associated with the object(s) designated by the *object* and * parameters. A style sheet modifies the font, font size and (except for automatic style sheets) font style.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In the *styleSheetName* parameter, you pass the name of the style sheet to be applied to the *object*. You can also pass either:

- the name of an existing style sheet (if the style sheet does not exist, an error is returned, that you can intercept using a method installed by the **ON ERR CALL** command),
- an empty string ("") so as to not apply the style sheet to the *object*, or
- one of the following constants found in the "**Font Styles**" theme in order to apply an automatic style sheet:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Automatic style sheet | String | __automatic__ | Used by default for all objects |
| Automatic style sheet_additional | String | __automatic_additional_text__ | Supported by static text, fields and variables only. Used for additional text in dialog boxes. |
| Automatic style sheet_main | String | __automatic_main_text__ | Supported by static text, fields and variables only. Used for main text in dialog boxes. |

If a style sheet was already associated with the object in Design mode, calling this command replaces it for the current process.

During the session, if you use the **ST SET ATTRIBUTES**, **ST SET TEXT** or **OBJECT SET FONT** commands on the *object* in order to modify its font or font size, the reference to the style sheet is automatically deleted from the object -- even if you assign the same attributes as those of the style sheet. However, if you modify the style (bold, italic, etc.), for example using the **ST SET ATTRIBUTES** or **OBJECT SET FONT STYLE** commands, these new properties are added to the style sheet for the duration of the session.

# OBJECT SET SUBFORM

OBJECT SET SUBFORM ( {* ;} object {; aTable}; detailSubform {; listSubform} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| aTable | Table | ⇒ | Table of form (if table form) |
| detailSubform | Text | ⇒ | Name of detail form of subform |
| listSubform | Text | ⇒ | Name of list form of subform (table form) |

## Description

The **OBJECT SET SUBFORM** command dynamically modifies the detail form as well as, optionally, the screen list form associated with the subform object designated by the *object* and * parameters.

**Note:** This command cannot change the type of the subform itself (list or page). This property can only be set in Design mode.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

In the *aTable* parameter, pass the table of the forms to be used. This parameter is optional; you can omit it when you specify a project form as detail subform.

In the *detailSubform* parameter, pass the name of the form to use as detail subform.

In the *listSubform* parameter, pass the name of the form to use as list subform. This parameter can only be passed when you modify a list type subform.

When you modify a page subform, the command can be executed at any time; current selections are not modified. However, when you modify a list subform, it can only be modified when it displays the list. If the command is executed when the detail form is displayed following a double-click in the list, an error is generated.

# OBJECT SET TEXT ORIENTATION

OBJECT SET TEXT ORIENTATION ( {* ;} object ; orientation )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| orientation | Longint | ⇒ | Value of object orientation |

## Description

The **OBJECT SET TEXT ORIENTATION** command modifies the orientation of the object(s) designated by the *object* and * parameters for the current process.

The "Orientation" property, available in the Form editor, performs permanent rotations of text areas in your forms. Unlike this property, the **OBJECT SET TEXT ORIENTATION** command applies the rotation to the contents of the object, but not to the object itself. For more information, refer to the *Design Reference* manual.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).
Only static text as well as non-enterable variables and fields can be rotated. If you apply this command to an object that does not support text orientation, the command does nothing.

In the *orientation* parameter, you pass the absolute orientation that you want to assign to the object. You must use one of the following constants, found in the "**Form Objects (Properties)**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Orientation 0° | Longint | 0 | No rotation (default value) |
| Orientation 180° | Longint | 180 | Orientation of text to 180° clockwise |
| Orientation 90° left | Longint | 270 | Orientation of text to 90° counter-clockwise |
| Orientation 90° right | Longint | 90 | Orientation of text to 90° clockwise |

**Note:** Only angles corresponding to these values are supported. If you pass any other value, it will be ignored.

## Example

You want to apply an orientation of 270° to a variable in your form:

```
OBJECT SET ENTERABLE(*;"myVar";False)
 // mandatory if variable is enterable
OBJECT SET TEXT ORIENTATION(*;"myVar";Orientation 90°  left)
```

# OBJECT SET THREE STATES CHECKBOX

OBJECT SET THREE STATES CHECKBOX ( {* ;} object ; threeStates )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| threeStates | Boolean | ⇒ | True = three-states checkbox, False = standard checkbox |

## Description

The **OBJECT SET THREE STATES CHECKBOX** command modifies, for the current process, the "Three-States" property of the checkbox(es) designated by the *object* and * parameters.

The "Three-states" option allows the additional "semi-checked" state to be used for checkboxes. For more information, refer to **Three-States check box** in the *Design Reference* manual.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

This command only applies to checkboxes associated with variables, and not to Boolean fields that are represented as checkboxes.

In the *threeStates* parameter, pass **True** to enable the "three states" mode, or **False** to disable it.

# OBJECT SET TITLE

OBJECT SET TITLE ( {* ;} object ; title )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an Object Name (String) If omitted, object is a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Variable (if * is omitted) |
| title | String | ⇒ | New title for the object |

## Description

The **OBJECT SET TITLE** command changes the title of the object(s) specified by *object* to the value you pass in *title*.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section **Object Properties**.

**OBJECT SET TITLE** can be applied to any simple objects that display a title:

- buttons and 3D buttons,
- check boxes and 3D check boxes,
- radio buttons and 3D radio buttons,
- listbox headers,
- static text areas,
- group boxes.

Usually, you will apply this command to one object at a time. The object title area must be big enough to hold the text; otherwise, the text is truncated.

Do not use carriage returns in *title*.

If you want to set a title on more than one line, use the "¥" character ("¥¥" in the code editor) as a line return. This is permitted for 3D buttons, 3D check boxes, 3D radio buttons and list box headers.

**Note:** Pass "¥¥¥" when you want to use the "¥" character in the title.

## Example 1

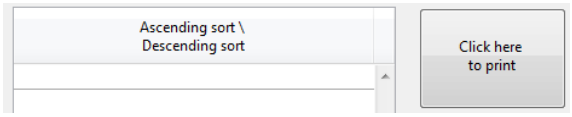
The following example is the object method of a search button located in the footer area of an output form displayed using **MODIFY SELECTION**. The method searches a table; depending on the search results, it enables or disables a button labeled *bDelete* and changes its title:

```
QUERY([People];[People]Name=vName)
Case of
   :(Records in selection([People])=0) // No people found
      OBJECT SET TITLE(bDelete;" Delete")
      OBJECT SET ENABLED(bDelete;False)
   :(Records in selection([People])=1) // One person found
      OBJECT SET TITLE(bDelete;"Delete Person")
      OBJECT SET ENABLED(bDelete;True)
   :(Records in selection([People])>1) // Many people found
      OBJECT SET TITLE(bDelete;"Delete People")
      OBJECT SET ENABLED(bDelete;True)
End case
```

## Example 2

You want to insert titles on two lines:

```
OBJECT SET TITLE(*;"header1";"Ascending sort ¥¥¥ ¥¥Descending sort")
OBJECT SET TITLE(*;"button1";"Click here ¥¥to print")
```


Ascending sort \
Descending sort
Click here
to print

# OBJECT SET VERTICAL ALIGNMENT

OBJECT SET VERTICAL ALIGNMENT ( {* ;} object ; alignment )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string)<br>If omitted, object is a variable |
| object | Form object | ⇒ | Object Name (if * is specified) or<br>Variable (if * is omitted) |
| alignment | Longint | ⇒ | Alignment code |

## Description

The **OBJECT SET VERTICAL ALIGNMENT** command modifies the type of vertical alignment applied to the object(s) designated by the *object* and * parameters.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (a string). If you do not pass this parameter, this indicates that the *object* is a variable. In this case, you pass a variable reference instead of a string.

In *alignment*, you pass one of the following constants found in the **Form Objects (Properties)** theme:

| Constant | Type | Value |
|---|---|---|
| Align bottom | Longint | 4 |
| Align center | Longint | 3 |
| Align default | Longint | 1 |
| Align top | Longint | 2 |

Vertical alignment can be applied to the following form objects:

- list boxes,
- list box columns,
- list box headers and footers.

# OBJECT SET VISIBLE

OBJECT SET VISIBLE ( {* ;} object ; visible )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, Object is an Object Name (String) If omitted, Object parameter is a Field or a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Field or Variable (if * is omitted) |
| visible | Boolean | ⇒ | True for visible, False for invisible |

## Description

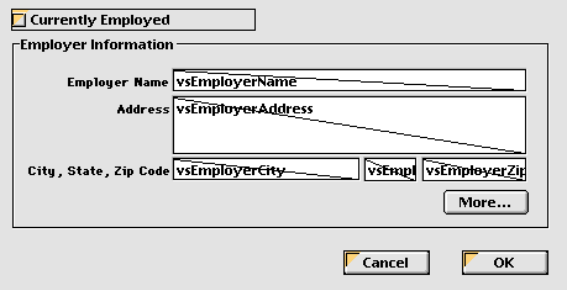
The **OBJECT SET VISIBLE** command shows or hides the objects specified by *object*.

If you specify the optional * parameter, you indicate an object name (a string) in *object*. If you omit the optional * parameter, you indicate a field or a variable in *object*. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section **Object Properties**.

If you pass *visible* equal to **TRUE**, the objects are shown. If you pass *visible* equal to **FALSE**, the objects are hidden.

## Example

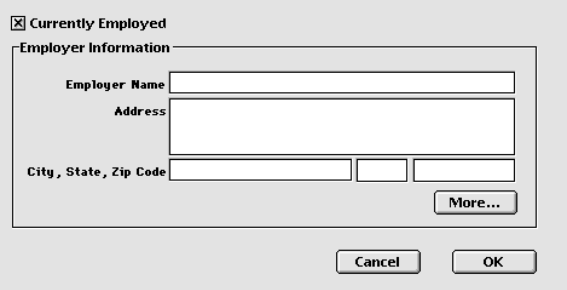
Here is a typical form in the Design environment:




The objects in the **Employer Information** group box each have an object name that contains the expression “employer” (including the group box). When the **Currently Employed** check box is checked, the objects must be visible; when the check box is unchecked, the objects must be invisible.
Here is the object method of the check box:

```
  ` cbCurrentlyEmployed Check Box Object Method
Case of
   :(Form event=On Load)
      cbCurrentlyEmployed:=1

   :(Form event=On Clicked)
  ` Hide or Show all the objects whose name contains "emp"
      OBJECT SET VISIBLE(*;"@emp@";cbCurrentlyEmployed#0)
  ` But always keep the check box itself visible
      OBJECT SET VISIBLE(cbCurrentlyEmployed;True)
End case
```

Therefore, when executed, the form looks like:




or:


Currently Employed
Cancel
OK

# _o_DISABLE BUTTON

_o_DISABLE BUTTON ( {* ;} object )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an Object Name (String) If omitted, object is a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Variable (if * is omitted) |

## Compatibility note

The **_o_DISABLE BUTTON** command was declared obsolete in 4D since version 12 and is kept only for compatibility reasons. Its overall scope, which includes all instances of the designated variable and not only those of the current form, does not correspond to those of the "Objects (Forms)" command theme.
**_o_DISABLE BUTTON** can be replaced favorably by the **OBJECT SET ENABLED** command.

# _o_ENABLE BUTTON

_o_ENABLE BUTTON ( {* ;} object )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an Object Name (String) If omitted, object is a Variable |
| object | Form object | ⇒ | Object Name (if * is specified), or Variable (if * is omitted) |

## Compatibility note

The **_o_ENABLE BUTTON** command is declared obsolete in 4D beginning with version 12 and is kept only for compatibility reasons. Its overall scope, which includes all instances of the designated variable and not only those of the current form, does not correspond to those of the "Object (Forms)" command theme.
**_o_ENABLE BUTTON** and **_o_DISABLE BUTTON** can be replaced favorably by the **OBJECT SET ENABLED** and **OBJECT Get enabled** commands.

# Objects (Language)

- Structure of 4D language objects
- OB Copy
- OB Get
- OB GET ARRAY
- OB GET PROPERTY NAMES
- OB Get type
- OB Is defined
- OB Is empty
- OB REMOVE
- OB SET
- OB SET ARRAY
- OB SET NULL

## Structure of 4D language objects

---

The commands of the **Objects (Language)** theme create and work with data in object form. This extends the exchange opportunities between 4D and any application that supports structured data.

All the commands in this theme support the following 4D objects:

- Object variables or object arrays created and initialized using the **C_OBJECT** ("**Compiler**" theme) or **ARRAY OBJECT** ("**Arrays**" theme) commands.
- Object fields from the 4D database (see **4D field types**).

The structure of "native" 4D objects is based on the classic principle of "property/value" pairs. The syntax of these objects is based on JSON notation, but does not follow it completely.

**Note:** To work with JSON objects, you must use the commands found in the "**JSON**" theme.

- An attribute **name** is always a text, for example "Name".
- An attribute **value** can be one of the following types:
  - number (Real, Integer, etc.)
  - text
  - array (text, real, longint, integer, Boolean, object, pointer)
  - null
  - Boolean
  - date (format "YYYY-MM-DDTHH:mm:ss.SSSZ")
  - object (objects can be nested on several levels)
  - object variables and arrays also support pointers (stored as such, evaluated using the **JSON Stringify** command or when copying).

**Warning:** Keep in mind that attribute names differentiate between upper and lower case.

# OB Copy

OB Copy ( object {; resolvePtrs} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ➡ | Structured object |
| resolvePtrs | Boolean | ➡ | True = resolve pointers, False or omitted = do not resolve pointers |
| Function result | Object | ⮌ | Copy of object |

## Description

The **OB Copy** command returns an object containing a complete copy of the properties, sub-objects and values for the *object*.

*object* must have been defined using the **C_OBJECT** command or designate a 4D object field.

If *object* contains pointer type values, by default the copy also contains the pointers. However, you can resolve pointers when copying by passing **True** in the *resolvePtrs* parameter. In this case, each pointer present as a value in object is evaluated when copying and its dereferenced value is used.

## Example 1

You want to duplicate an object containing simple values:

```
C_OBJECT($Object)
C_TEXT($JsonString)

ARRAY OBJECT($arraySel;0)
ALL RECORDS([Product])
While(Not(End selection([Product])))
   OB SET($Object;"id";[Product]ID_Product)
   OB SET($Object;"Product Name";[Product]Product_Name)
   OB SET($Object;"Price";[Product]Price)
   OB SET($Object;"Tax rate";[Product]Tax_rate)
   $ref_value:=OB Copy($Object) //direct copy
   APPEND TO ARRAY($arraySel;$ref_value)
 //the contents of $ref_value are identical to those of $Object
   NEXT RECORD([Product])
End while
 //the contents of $ref_value
$JsonString:=JSON Stringify array($arraySel)
```

## Example 2

You duplicate an object containing pointers:

```
C_OBJECT($ref)

OB SET($ref;"name";->[Company]name) //object with pointers
OB SET($ref;"country";->[Company]country)
ARRAY OBJECT($sel;0)
ARRAY OBJECT($sel2;0)

ALL RECORDS([Company])

While(Not(End selection([Company])))
   $ref_comp:=OB Copy($ref) // copy without evaluating pointers
 // $ref_comp={"name":"->[Company]name","country":"->[Company]Country"}
   $ref_comp2:=OB Copy($ref;True) //copy with evaluation of pointers
 // $ref_comp={"name":"4D SAS","country":"France"}
   APPEND TO ARRAY($sel;$ref_comp)
   APPEND TO ARRAY($sel2;$ref_comp2)
```

```
    NEXT RECORD([Company])
End while

$Object:=JSON Stringify array($sel)
$Object2:=JSON Stringify array($sel2)

 // $Object = [{"name":"","country":""},{"name":"","country":""},...]
 // $Object2 = [{"name":"4D SAS","country":"France"},{"name":"4D, Inc","country":"USA"},
{"name":"Catalan","country":"France"}...]
```

# OB Get

OB Get ( object ; property {; type} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ➡ | Structured object |
| property | Text | ➡ | Name of property to read |
| type | Longint | ➡ | Type to which to convert the value |
| Function result | Boolean, Date, Object, Pointer, Real, Text | ↩ | Current value of property |

## Description

The **OB Get** command returns the current value of the *property* of the *object*, optionally converted into the *type* specified.

*object* must have been defined using the **C_OBJECT** command or designate a 4D object field.

**Note:** This command supports attribute definitions in 4D Write Pro *objects*, like the **WP GET ATTRIBUTES** command (see example 9). However, unlike **WP GET ATTRIBUTES**, **OB Get** does not allow you to handle a picture variable or field directly as an attribute value.

In the *property* parameter, pass the label of the property to be read. Note that the *property* parameter is case sensitive.

By default, 4D returns the value of the property in its original type. You can "force" the typing of the value returned using the optional *type* parameter. To do this, in *type* you pass one of the following constants found in the **Field and Variable Types** theme:

| Constant | Type | Value |
|---|---|---|
| Is Boolean | Longint | 6 |
| Is date | Longint | 4 |
| Is longint | Longint | 9 |
| Is object | Longint | 38 |
| Is pointer | Longint | 23 |
| Is real | Longint | 1 |
| Is text | Longint | 2 |
| Is time | Longint | 11 |

The command returns the value of the *property*. Several types of data are supported. Note that:

- a pointer is returned as such; it can be evaluated using the **JSON Stringify** command,
- dates are returned in the format "YYYY-MM-DDTHH:mm:ss.SSSZ"
- in real values, the decimal separator is always a period "."
- times are returned as a number. Note that **OB SET** stores times as milliseconds, in compliance with the JavaScript standard, while 4D expects a number of seconds. In order for **OB Get** to interpret a stored time correctly, you need to use the Is time constant.

## Example 1

Retrieving a text type value:

```
C_OBJECT($ref)
C_TEXT($FirstName)
OB SET($ref;"FirstName";"Harry")
$FirstName:=OB Get($ref;"FirstName") // $FirstName = "Harry" (text)
```

## Example 2

Retrieving a real number value converted into a longint:

```
OB SET($ref ;"age";42)
$age:=OB Get($ref ;"age") // $age is a real number (default)
$age:=OB Get($ref ;"age";Is longint) // $age is a longint
```

## Example 3

Retrieving the values of an object:

```
C_OBJECT($ref1;$ref2)
OB SET($ref1;"LastName";"Smith") // $ref1={"LastName":"Smith"}
OB SET($ref2;"son";$ref1) // $ref2={"son":{"LastName":"Smith"}}
$son:=OB Get($ref2;"son") // $son={"LastName":"john"} (object)
$sonsName:=OB Get($son ;"name") // $sonsName="john" (text)
```

## Example 4

Modifying the age of an employee twice:

```
C_OBJECT($ref_john;$ref_jim)
OB SET($ref_john;"name";"John";"age";35)
OB SET($ref_jim;"name";"Jim";"age";40)
APPEND TO ARRAY($myArray;$ref_john) // we create an object array
APPEND TO ARRAY($myArray;$ref_jim)
 // we change the age for John from 35 to 25
OB SET($myArray{1};"age";25)
 // We replace the age of "John" in the array
For($i;1;Size of array($myArray))
   If(OB Get($myArray{$i};"name")="John")
      OB SET($myArray{$i};"age";36) // instead of 25
 // $ref_john={"name":"John","age":36}
   End if
End for
```

## Example 5

Deserializing a data string formatted in ISO:

```
C_OBJECT($object)
C_DATE($birthday)
C_TEXT($birthdayString)
OB SET($object;"Birthday";"1990-12-25T12:00:00Z")
$birthdayString:=OB Get($object;"Birthday")
 // $birthdayString="1990-12-25T12:00:00Z"
$birthday:=OB Get($object;"Birthday";Is date)
 // $birthday=25/12/90
```

## Example 6

Using nested objects:

```
C_OBJECT($ref1;$child;$children)
C_TEXT($childName)
OB SET($ref1;"firstname";"John";"lastname";"Monroe")
 //{"firstname":"john","lastname":"Monroe"}
OB SET($children;"children";$ref1)
$child:=OB Get($children;"children")
 //$son = {"firstname":"John","lastname":"Monroe"} (object)
$childName:=OB Get($child;"lastname")
 //$childName = "Monroe" (text)
 //or
$childName:=OB Get(OB Get($children;"children");"lastname")
 // $childName = "Monroe" (text)
```

## Example 7

Recovery in 4D of a time stored in an object:

```
C_OBJECT($obj_o)
C_TIME($set_h;$get_h)

$set_h:=?01:00:00?+1
OB SET($obj_o;"myHour";$set_h)
 // $obj_o == {"myHour":3601000}
 // The time is stored in milliseconds
$get_h:=OB Get($obj_o;"myHour";Is time)
 // $get_h == ?01:00:01?
 // The time is read correctly
```

## Example 8

Examples of working with 4D object fields:

```
 // Define a value
OB SET([People]Identity_OB;"First name";$firstName)
OB SET([People]Identity_OB;"Last name";$lastName)

 // Get a value
$firstName:=OB Get([People]Identity_OB;"First name")
$lastName:=OB Get([People]Identity_OB;"Last name")
```

## Example 9

In the method of a form containing a 4D Write Pro area, you can write:

```
If(Form event=On Validate)
   OB SET([MyDocuments]My4DWP;"myatt_Last edition by";Current user)
   OB SET([MyDocuments]My4DWP;"myatt_Category";"Memo")
End if
```

You can also read custom attributes of the documents:

```
vAttrib:=OB Get([MyDocuments]My4DWP;"myatt_Last edition by")
```

# OB GET ARRAY

OB GET ARRAY ( object ; property ; array )

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ➡ | Structured object |
| property | Text | ➡ | Name of property to read |
| array | Text array, Real array, Boolean array, Object array, Pointer array, Longint variable | ⬅ | Value array of property |

## Description

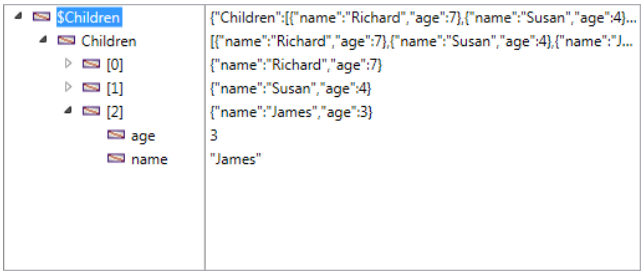
The **OB GET ARRAY** command retrieves, in *array*, the array of values stored in the *property* of the language object designated by the *object* parameter.

*object* must have been defined using the **C_OBJECT** command or designate a 4D object field.

In the *property* parameter, pass the label of the property to be read. Note that the *property* parameter is case sensitive.

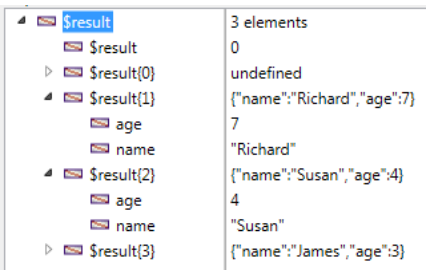
## Example 1

Given an object array defined in the example of the **OB SET ARRAY** command:




We want to retrieve these values:

```
ARRAY OBJECT($result;0)
OB GET ARRAY($Children;"Children";$result)
```




## Example 2

We want to change a value in the first element of the array:

```
 // Change the value of "age":
ARRAY OBJECT($refs)
OB GET ARRAY($refEmployees;"__ENTITIES";$refs)
OB SET($refs{1};"age";25)
```

# OB GET PROPERTY NAMES

OB GET PROPERTY NAMES ( object ; arrProperties {; arrTypes} )

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ⇨ | Structured object |
| arrProperties | Text array | ⇦ | Property names |
| arrTypes | Longint array | ⇦ | Property types |

## Description

The **OB GET PROPERTY NAMES** command returns, in *arrProperties*, the names of the properties contained in the language object designated by the *object* parameter.

*object* must have been defined using the **C_OBJECT** command or designate a 4D object field.

Pass a text array in the *arrProperties* parameter. If the array does not exist, the command creates and sizes it automatically.

Optionally, you can also pass a longint array in *arrTypes*. For each element of *arrProperties*, the command returns, in *arrTypes*, the type of value stored in the property. You can compare the values received with the following constants, found in the "**Field and Variable Types**" theme:

| Constant | Type | Value |
|---|---|---|
| Is Boolean | Longint | 6 |
| Is JSON null | Longint | 255 |
| Is object | Longint | 38 |
| Is real | Longint | 1 |
| Is text | Longint | 2 |
| Object array | Longint | 39 |

## Example 1

You want to test that an object is not empty:

```
ARRAY TEXT(arrNames;0)
ARRAY LONGINT(arrTypes;0)
C_OBJECT($ref_richard)
OB SET($ref_richard;"name";"Richard";"age";7)
OB GET PROPERTY NAMES($ref_richard;arrNames;arrTypes)
 // arrNames{1}="name", arrNames{2}="age"
 // arrTypes{1}=2, arrTypes{2}=1
If(Size of array(arrNames)#0)
 // ...
End if
```

## Example 2

Using an object array element:

```
C_OBJECT($Children;$ref_richard;$ref_susan;$ref_james)
ARRAY OBJECT($arrayChildren;0)

OB SET($ref_richard;"name";"Richard";"age";7)
APPEND TO ARRAY($arrayChildren;$ref_richard)
OB SET($ref_susan;"name";"Susan";"age";4;"girl";True) //additional attribute
APPEND TO ARRAY($arrayChildren;$ref_susan)
OB SET($ref_james;"name";"James")
OB SET NULL($ref_james;"age") //null attribute
APPEND TO ARRAY($arrayChildren;$ref_james)

OB GET PROPERTY NAMES($arrayChildren{1};$arrNames;$arrTypes)
```

```
 // $arrayChildren{1} = {"name":"Richard","age":7}
 // $arrNames{1}="name"
 // $arrNames{2}="age"
 // $arrTypes{1}=2
 // $arrTypes{2}=1

OB GET PROPERTY NAMES($arrayChildren{2};$arrNames;$arrTypes)
 // $arrayChildren{3} = {"name":"Susan","age":4,"girl":true}
 // $arrNames{1}="name"
 // $arrNames{2}="age"
 // $arrNames{3}="girl"
 // $arrTypes{1}=2
 // $arrTypes{2}=1
 // $arrTypes{3}=6

OB GET PROPERTY NAMES($arrayChildren{3};$arrNames;$arrTypes)
 // $arrayChildren{3} = {"name":"James","age":null}
 // $arrNames{1}="name"
 // $arrNames{2}="age"
 // $arrTypes{1}=2
 // $arrTypes{2}=255
```

# OB Get type

OB Get type ( object ; property ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ➡ | Structured object |
| property | Text | ➡ | Property name |
| Function result | Longint | ⮌ | Property value type |

## Description

The **OB Get type** command returns the type of value associated with the *property* of the language *object*.

*object* must have been defined using the **C_OBJECT** command or designate a 4D object field.

In the *property* parameter, pass the label of the property whose type you want to find out.
Note that the *property* parameter is case sensitive.

The command returns a longint indicating the type of value. You can compare this value with the following constants, found in the "**Field and Variable Types**" theme:

| Constant | Type | Value |
|---|---|---|
| Is Boolean | Longint | 6 |
| Is JSON null | Longint | 255 |
| Is object | Longint | 38 |
| Is real | Longint | 1 |
| Is text | Longint | 2 |
| Is undefined | Longint | 5 |
| Object array | Longint | 39 |

## Example

We want to get the type of standard values:

```
C_OBJECT($ref)
OB SET($ref;"name";"smith";"age";42)
$type:=OB Get type($ref;"name") //$type returns 2
$type2:=OB Get type($ref;"age") //$type2 returns 1
```

# OB Is defined

OB Is defined ( object {; property} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ➡ | Structured object |
| property | Text | ➡ | If passed = property to check, if omitted = check object |
| Function result | Boolean | ⮌ | If property omitted: True if object is defined, otherwise False.<br>If property passed: True if property is defined, otherwise False |

## Description

The **OB Is defined** command returns True if *object* or *property* is defined, and False otherwise.

*object* must have been created using the **C_OBJECT** command or designate a 4D object field.

By default, if you omit the *property* parameter, the command checks whether the *object* is defined. An object is defined if its contents has been initialized.

**Note:** An object can be defined but empty. To find out if an object is undefined or empty, use the **OB Is empty** command.

If you pass the *property* parameter, the command checks whether this property exists in *object*. Note that the *property* parameter is case sensitive.

## Example 1

Syntax testing the initialization of an object:

```
C_OBJECT($object)
$def:=OB Is defined($object) // $def=false since $object is not initialized

OB SET($object;"Name";"Martin")
OB REMOVE($object;"Name")
$def2:=OB Is defined($object) // $def2=true since $object is empty {} but has been initialized
```

## Example 2

Test for existence of a property:

```
C_OBJECT($ref)
OB SET($ref;"name";"smith";"age";42)
 //...
If(OB Is defined($ref;"age"))
 //...
End if
```

This test is equivalent to:

```
If(OB Get type($Object;"name")#Is undefined)
```

# OB Is empty

OB Is empty ( object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ⇒ | Structured object |
| Function result | Boolean | ⇐ | True if object is empty or undefined, otherwise False |

## Description

The **OB Is empty** command returns True if *object* is undefined or empty, and False if *object* is defined (initialized) and contains at least one property.

*object* must have been created using the **C_OBJECT** command or designate a 4D object field.

## Example

Here are the different results of this command as well as the **OB Is defined** command, depending on the context:

```
C_OBJECT($ref)
$empty:=OB Is empty($ref) // True
$def:=OB Is defined($ref) // False

OB SET($ref;"name";"Susie";"age";4)
 // $ref="{"name":"Susie","age":4}"
$empty:=OB Is empty($ref) // False
$def:=OB Is defined($ref) // True

OB REMOVE($ref;"name")
OB REMOVE($ref;"age")
$empty:=OB Is empty($ref) // True
$def:=OB Is defined($ref) // True
```

# OB REMOVE

OB REMOVE ( object ; property )

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ⇒ | Structured object |
| property | Text | ⇒ | Name of property to remove |

## Description

The **OB REMOVE** command removes the *property* of the language object designated by the *object* parameter. This command removes the *property* as well as its current value.

*object* must have been defined using the **C_OBJECT** command or designate a 4D object field.

In the *property* parameter, pass the label of the property to be read. Note that the *property* parameter is case sensitive.

## Example

You want to remove the "age" property of an object:

```
C_OBJECT($Object)
OB SET($Object;"name";"smith";"age";42;"client";True)
 // $Object={"name":"smith","age":42,"client":true}
OB REMOVE($Object;"age")
 // $Object={"name":"smith","client":true}
```

# OB SET

OB SET ( object ; property ; value {; property2 ; value2 ; ... ; propertyN ; valueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ⇒ | Structured object |
| property | Text | ⇒ | Name of property to set |
| value | Text, Date, Boolean, Pointer, Number, Object | ⇒ | New value of property |

## Description

The **OB SET** command creates or modifies one or more *property*/*value* pairs in the language object designated by the *object* parameter.

*object* must have been defined using the **C_OBJECT** command or designate a 4D object field.

**Note:** This command supports attribute definitions in 4D Write Pro *objects*, like the **WP SET ATTRIBUTES** command (see example 10). However, unlike **WP SET ATTRIBUTES**, **OB SET** does not allow you to handle a picture variable or field directly as an attribute value.

In the *property* parameter, pass the label of the property to be created or modified. If the property already exists in *object*, its value is updated. If it does not exist, it is created.
Note that the *property* parameter is case sensitive.

In the *value* parameter, pass the value you want to set for the property. Several data types are supported. Note that:

- if you pass a pointer, it is kept as is; it is evaluated using the **JSON Stringify** command,
- dates are stored in the format "YYYY-MM-DDTHH:mm:ss.SSSZ". When converting 4D dates into text prior to storing them in the object, by default the program takes the local time zone into account. You can modify this behavior using the JSON use local time selector of the **SET DATABASE PARAMETER** command.
- if you pass a time, it is stored as a number of milliseconds (Real) in *object*.
- if you pass a language object, the command uses the object reference and not a copy.

## Example 1

Creating an object and adding a text type property:

```
C_OBJECT($Object)
OB SET($Object ;"FirstName";"John";"LastName";"Smith")
 // $Object = {"FirstName":"John","LastName":"Smith"}
```

## Example 2

Creating an object and adding a Boolean type property:

```
C_OBJECT($Object)
OB SET($Object ;"LastName";"smith";"age";42;"client";True)
 // $Object = {"LastName":"smith","age":42,"client":true}
```

## Example 3

Modifying a property:

```
 // $Object = {"FirstName":"John","LastName":"Smith"}
OB SET($Object ;"FirstName";"Paul")
 // $Object = {"FirstName":"Paul","LastName":"Smith"}
```

## Example 4

Adding a property:

```
 // $Object = {"FirstName":"John","LastName":"Smith"}
OB SET($Object ;"department";"Accounting")
 // $Object = {"FirstName":"Paul","LastName":"Smith","department":"Accounting"}
```

## Example 5

Renaming a property:

```
C_OBJECT($Object)
OB SET($Object ;"LastName";"James";"age";35)
 // $Object = {"LastName":"James","age":35}
OB SET($Object ;"FirstName";OB Get($Object ;"LastName"))
 // $Object = {"FirstName":""James","nom":"James","age":35}
OB REMOVE($Object ;"LastName")
 // $Object = {"FirstName":""James","age":35}
```

## Example 6

Using a pointer:

```
 // $Object = {"FirstName":"Paul","LastName":"Smith"}
C_TEXT($LastName)
OB SET($Object ;"LastName";->$LastName)
 // $Object = {"FirstName":"Paul","LastName":"->$LastName"}
$JsonString:=JSON Stringify($Object)
 // $JsonString="{"FirstName":"Paul","LastName":""}
$LastName:="Wesson"
$JsonString:=JSON Stringify($Object)
 // $JsonString="{"FirstName":"Paul","LastName":"Wesson"}
```

## Example 7

Using an object:

```
C_OBJECT($ref_smith)
OB SET($ref_smith ;"name";"Smith")
C_OBJECT($ref_emp)
OB SET($ref_emp ;"employee";$ref_smith)
$Json_string :=JSON Stringify($ref_emp)
 // $ref_emp = {"employee":{"name":"Smith"}} (object)
 // $Json_string = "{"employee":{"name":"Smith"}}" (string)
```

You can also change a value on the fly:

```
OB SET($ref_smith ;"name";"Smyth")
 // $ref_smith = {"employee":{"name":"Smyth"}}
$string:=JSON Stringify($ref_emp)
 // $string = "{"employee":{"name":"Smyth"}}"
```

## Example 8

If you have defined the [Rect]Desc field as an object field, you can write:

```
CREATE RECORD([Rect])
[Rect]Name:="Blue square"
OB SET([Rect]Desc;"x";"50";"y";"50";"color";"blue")
SAVE RECORD([Rect])
```

## Example 9

You want to export data in JSON that contains a converted 4D date. Note that conversion occurs when the date is saved in the object, so you must call the **SET DATABASE PARAMETER** command before calling **OB SET**:

```
C_OBJECT($o)
SET DATABASE PARAMETER(JSON use local time;0)
OB SET($o ;"myDate";Current date) // JSON conversion
$json:=JSON Stringify($o)
SET DATABASE PARAMETER(JSON use local time;1)
```

## Example 10

In the method of a form containing a 4D Write Pro area, you can write:

```
If(Form event=On Validate)
   OB SET([MyDocuments]My4DWP;"myatt_Last edition by";Current user)
   OB SET([MyDocuments]My4DWP;"myatt_Category";"Memo")
End if
```

You can also read custom attributes of the documents:

```
vAttrib:=OB Get([MyDocuments]My4DWP;"myatt_Last edition by")
```

# OB SET ARRAY

OB SET ARRAY ( object ; property ; array )

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ➡ | Structured object |
| property | Text | ➡ | Name of property to set |
| array | Text array, Real array, Boolean array, Object array, Pointer array, Longint array | ➡ | Array to store in property |

## Description

The **OB SET ARRAY** command defines the *array* to be associated with the *property* in the language object designated by the *object* parameter.

*object* must have been defined using the **C_OBJECT** command or designate a 4D object field.

In the *property* parameter, pass the label of the property to be created or modified. If the property already exists in *object*, its value is updated. If it does not exist, it is created.
Note that the *property* parameter is case sensitive.

In the *array* parameter, pass the array that must be passed as the property value. Several array types are supported.

**Note:** It is not possible to use two-dimensional arrays.

## Example 1

Using a text array:

```
C_OBJECT($Children)
ARRAY TEXT($arrChildren;3)
$arrChildren{1}:="Richard"
$arrChildren{2}:="Susan"
$arrChildren{3}:="James"

OB SET ARRAY($Children;"Children";$arrChildren)
 // Value of $Children = {"Children":["Richard","Susan","James"]}
```

## Example 2

Adding an element to an array:

```
ARRAY TEXT($arrText;2)
$arrText{1}:="Smith"
$arrText{2}:="White"
C_OBJECT($Employees)
OB SET ARRAY($Employees;"Employees";$arrText)
APPEND TO ARRAY($arrText;"Brown") // Add to the 4D array
 // $Employees = {"Employees":["Smith","White"]}

OB SET ARRAY($Employees;"Employees";$arrText)
 // $Employees = {"Employees":["Smith","White","Brown"]}
```

## Example 3

Using a text array with selection of an element:

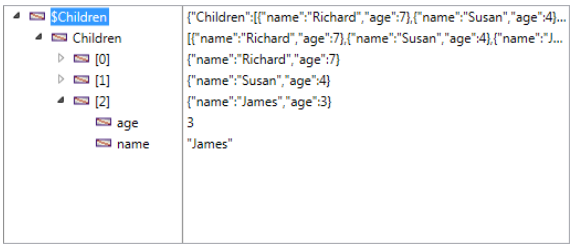
```
 // $Employees = {"Employees":["Smith","White","Brown"]}
OB SET ARRAY($Employees ;"Manager";$arrText{1})
 // $Employees = {"Employees":["Smith","White","Brown"],"Manager":["Smith"]}
```

## Example 4

Using an object array:

```
C_OBJECT($Children;$ref_richard;$ref_susan;$ref_james)
ARRAY OBJECT($arrChildren;0)
OB SET($ref_richard;"nom";"Richard";"age";7)
APPEND TO ARRAY($arrChildren;$ref_richard)
OB SET($ref_susan;"name";"Susan";"age";4)
APPEND TO ARRAY($arrChildren;$ref_susan)
OB SET($ref_james;"name";"James";"age";3)

APPEND TO ARRAY($arrChildren;$ref_james)

 // $arrChildren {1} = {"name":"Richard","age":7}
 // $arrChildren {2} = {"name":"Susan","age":4}
 // $arrChildren {3} = {"name":"James","age":3}

OB SET ARRAY($Children;"Children";$arrChildren)

 // $Children = {"Children":[{"name":"Richard","age":7},{"name":"Susan",
 // "age":4},{"name":"James","age":3}]}
```

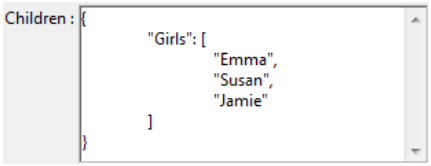
Here is how the object appears in the debugger:




## Example 5

Using an Object field:

```
ARRAY TEXT($arrGirls;3)
$arrGirls{1}:="Emma"
$arrGirls{2}:="Susan"
$arrGirls{3}:="Jamie"
OB SET ARRAY([People]Children;"Girls";$arrGirls)
```



# OB SET NULL

OB SET NULL ( object ; property )

| Parameter | Type | | Description |
|---|---|---|---|
| object | Object, Object Field | ⇒ | Structured object |
| property | Text | ⇒ | Name of property where null value is to be applied |

## Description

The **OB SET NULL** command stores the **null** value in the language object designated by the *object* parameter.

*object* must have been defined using the **C_OBJECT** command or designate a 4D object field.

In the *property* parameter, pass the label of the property where you want to store the **null** value. If the property already exists in *object*, its value is updated. If it does not exist, it is created.
Note that the *property* parameter is case sensitive.

## Example

We want to put the null value in the "age" property for Lea:

```
C_OBJECT($ref)
OB SET($ref;"name";"Lea";"age";4)
 // $ref = {"name":"Lea","age":4}
...
OB SET NULL($ref ;"age")
 // $ref = {"name":"Lea","age":null}
```

# On a Series

- On a Series
- Average Updated 16.0
- Max Updated 16.0
- Min Updated 16.0
- Std deviation
- Sum Updated 16.0
- Sum squares
- Variance

## On a Series

The functions of this theme perform calculations on a series of values.

The **Average**, **Max**, **Min**, **Sum**, **Sum squares**, **Std deviation** and **Variance** functions are applied to fields or arrays:

- When applied to fields, they use the current selection of records,
- When applied to arrays, they use array elements.

Note that when they are applied to fields, the **Sum squares**, **Std deviation** et **Variance** functions can be used only during printing.

All these functions work on numeric data only and return numeric values.

### Using statistical functions apart from printing

When **Average**, **Max**, **Min** or **Sum** are used on a field outside a printing operation, they may have to load each record in the current selection to calculate the result. If there are many records, this process may take some time. To limit the processing time, you can index the field.

**Note**: When the operation is long, a progress thermometer appears. This thermometer has a Stop button that lets the user interrupt the operation. If the user clicks this button, the OK variable is set to 0. If the operation is completed correctly, the OK variable is set to 1.

### Using statistical functions in a printed report

When statistical functions are used in a report, they behave in a specific way because the report itself must load each record. Use these functions in a form or object method when printing with the **PRINT SELECTION** command or when printing by choosing **Print** from the **File** menu in the Design environment.

When you use these functions in a report, the values that are returned are reliable only at break level 0, and only when break processing is turned on. This means that they are useful only at the end of a report, after all the records have been processed.

You would use these functions only in an object method for a non-enterable area that is included in the B0 Break area.

Remember that the field passed as a parameter to the statistical function must be a numeric.

# Average

Average ( series {; attributePath} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| series | Field, Array | ⇒ | Data for which to return the average |
| attributePath | Text | ⇒ | Path of attribute for which to return the average |
| Function result | Real | ⇐ | Arithmetic mean (average) of series |

## Description

**Average** returns the arithmetic mean (average) of *series*. If *series* is an indexed field, the index is used to find the average.

You can pass an array (one or two dimensions) in *series*. In this case, the array must be of the Integer, Longint or Real type.

This command accepts an optional *attributePath* parameter of the Text type, that you can use if *series* is an object field. It allows you to define the path of the attribute to compute. Use the standard dot notation to define paths to nested attributes, for example "company.address.number". Keep in mind that object attribute names are case-sensitive.
Only numeric attribute values are computed. If there are values in the attribute path which are not of a numeric type, they are ignored.

If the command is correctly executed, the OK system variable is set to 1. If it is interrupted (for example if the user clicks on the **Stop** button of the progress thermometer), the OK variable is set to 0.

## Example 1

The following example sets the variable *vAverage* that is in the B0 Break area of an output form. The line of code is the object method for *vAverage*. The object method is not executed until the level 0 break:

```
vAverage:=Average([Employees] Salary)
```

The following method is called to print the records in the selection and to activate break processing:

```
ALL RECORDS([Employees])
ORDER BY([Employees];[Employees]LastNm;>)
BREAK LEVEL(1)
ACCUMULATE([Employees]Salary)
FORM SET OUTPUT([Employees];"PrintForm")
PRINT SELECTION([Employees])
```

**Note:** The parameter to the **BREAK LEVEL** command should be equal to the number of breaks in your report. For more information about break processing, refer to the chapter **Printing**.

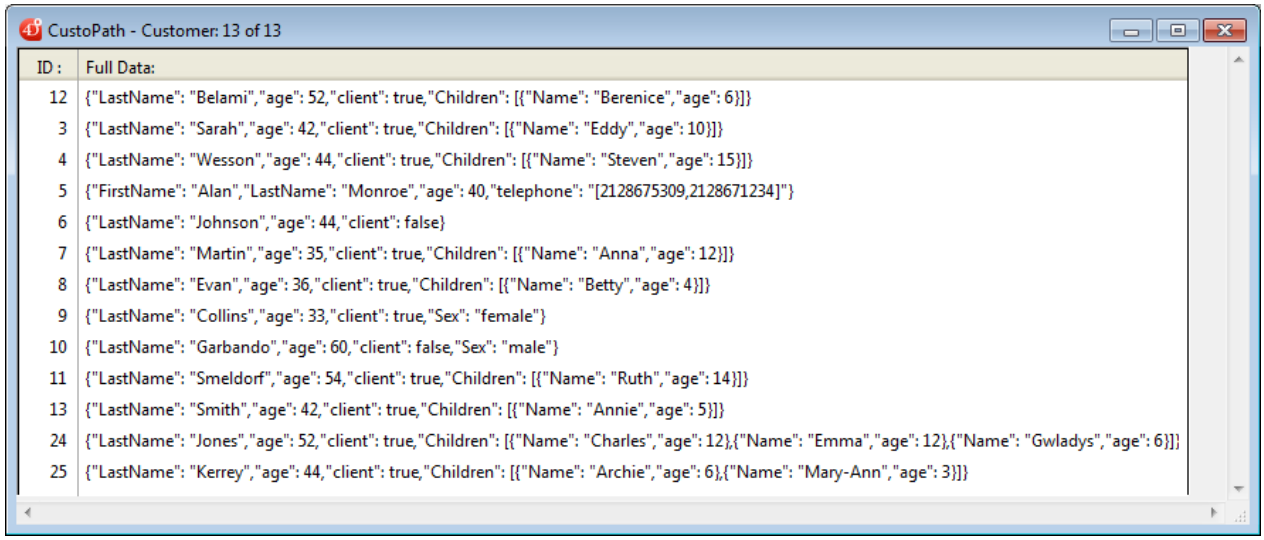
## Example 2

This example gets the average of the first 15 grades in the selection:

```
ARRAY REAL($ArrGrades;0)
QUERY([Exams];[Exams]Exam_Date=!01/07/11!)
ORDER BY([Exams];[Exams]Exam_Grade;<)
SELECTION TO ARRAY([Exams]Exam_Grade;$ArrGrades)
ARRAY REAL($ArrGrades;15)
vAverage:=Average($ArrGrades)
```

## Example 3

Your [Customer] table contains a "full_Data" object field with the following data:




You can perform the following computations:

```
C_REAL($vAvg)
ALL RECORDS([Customer])
$vAvg:=Average([Customer]full_Data;"age")
 //$vAvg is 44.46

C_LONGINT($vTot)
$vTot:=Sum([Customer]full_Data;"Children[].age")
 //$vTot is 105
```

# Max

Max ( series {; attributePath} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| series | Field, Array | ➡ | Data for which to return the maximum value |
| attributePath | Text | ➡ | Path of attribute for which to return the maximum value |
| Function result | Real | ⮌ | Maximum value in series |

## Description

**Max** returns the maximum value in *series*. If *series* is an indexed field, the index is used to find the maximum value.

You can pass an array (one or two dimensions) in *series*. In this case, the array must be of the Integer, Longint or Real type.

If the *series* selection is empty, **Max** returns 0.

This command accepts an optional *attributePath* parameter of the Text type, that you can use if *series* is an object field. It allows you to define the path of the attribute to compute. Use the standard dot notation to define paths to nested attributes, for example "company.address.number". Keep in mind that object attribute names are case-sensitive.
Only numeric attribute values are computed. If there are values in the attribute path which are not of a numeric type, they are ignored.

If the command is correctly executed, the OK system variable is set to 1. If it is interrupted (for example if the user clicks on the **Stop** button of the progress thermometer), the OK variable is set to 0.

## Example 1

The following example is an object method for the variable *vMax* placed in the break 0 portion of the form. The variable is printed at the end of the report. The object method assigns the maximum value of the field to the variable, which is then printed in the last break of the report.

```
vMax:=Max([Employees] Salary)
```

The following method is called to print the records in the selection and to activate break processing:

```
ALL RECORDS([Employees])
ORDER BY([Employees];[Employees]LastNm;>)
BREAK LEVEL(1)
ACCUMULATE([Employees]Salary)
FORM SET OUTPUT([Employees];"PrintForm")
PRINT SELECTION([Employees])
```

**Note:** The parameter to the **BREAK LEVEL** command should be equal to the number of breaks in your report. For more information about break processing, refer to the chapter **Printing**.

## Example 2

This example gets the highest value in the array:

```
ARRAY REAL($ArrGrades;0)
QUERY([Exams];[Exams]Exam_Date=!01/07/11!)
SELECTION TO ARRAY([Exams]Exam_Grade;$ArrGrades)
vMax:=Max($ArrGrades)
```

## Example 3

For an example of computing an object field attribute, please refer to the example 3 of the **Average** command description.

# Min

Min ( series {; attributePath} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| series | Field, Array | ⇒ | Data for which to return the minimum value |
| attributePath | Text | ⇒ | Path of attribute for which to return the minimum value |
| Function result | Real | ⇐ | Minimum value in series |

## Description

**Min** returns the minimum value in *series*. If *series* is an indexed field, the index is used to find the minimum value.

If the *series* selection is empty, **Min** returns 0.

You can pass an array (one or two dimensions) in *series*. In this case, the array must be of the Integer, Longint or Real type.

This command accepts an optional *attributePath* parameter of the Text type, that you can use if *series* is an object field. It allows you to define the path of the attribute to compute. Use the standard dot notation to define paths to nested attributes, for example "company.address.number". Keep in mind that object attribute names are case-sensitive.
Only numeric attribute values are computed. If there are values in the attribute path which are not of a numeric type, they are ignored.

If the command is correctly executed, the OK system variable is set to 1. If it is interrupted (for example if the user clicks on the **Stop** button of the progress thermometer), the OK variable is set to 0.

## Example 1

The following example is an object method for the variable *vMin* placed in the break 0 portion of the form. The variable is printed at the end of the report. The object method assigns the minimum value of the field to the variable, which is then printed in the last break of the report:

```
vMin:=Min([Employees]Salary)
```

The following method is called to print the records in the selection and to activate break processing:

```
ALL RECORDS([Employees])
ORDER BY([Employees];[Employees]LastNm;>)
BREAK LEVEL(1)
ACCUMULATE([Employees]Salary)
FORM SET OUTPUT([Employees];"PrintForm")
PRINT SELECTION([Employees])
```

**Note:** The parameter to the **BREAK LEVEL** command should be equal to the number of breaks in your report. For more information about break processing, refer to the chapter **Printing**.

## Example 2

The following example finds the lowest sale amount of an employee and displays the result in an alert box:

```
ALERT("Minimum sale = "+String(Min([Employees]Sales)))
```

## Example 3

This example gets the lowest value in the array:

```
ARRAY REAL($ArrGrades;0)
QUERY([Exams];[Exams]Exam_Date=!01/07/11!)
SELECTION TO ARRAY([Exams]Exam_Grade;$ArrGrades)
vMin:=Min($ArrGrades)
```

## Example 4

For an example of computing an object field attribute, please refer to the example 3 of the **Average** command description.

# Std deviation

Std deviation ( series ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| series | Field, Array | ⇒ | Data for which to return the standard deviation |
| Function result | Real | ⇐ | Standard deviation of series |

## Description

**Std deviation** returns the standard deviation of *series*. If *series* is an indexed field, the index is used to find the standard deviation.

You can pass an array (one or two dimensions) in *series*. In this case, the array must be of the Integer, Longint or Real type.

## Example 1

The following example is an object method for the variable *vDeviate*. The object method assigns the standard deviation for a data series to *vDeviate*:

```
vDeviate:=Std deviation([Table1]DataSeries)
```

The following method is called to print the records in the selection and to activate break processing:

```
ALL RECORDS([Table1])
ORDER BY([Table1];[Table1]DataSeries;>)
BREAK LEVEL(1)
ACCUMULATE([Table1]DataSeries)
OUTPUT FORM([Table1];"PrintForm")
PRINT SELECTION([Table1])
```

**Note:** The parameter to the **BREAK LEVEL** command should be equal to the number of breaks in your report. For more information about break processing, refer to the printing commands.

## Example 2

This example gets the standard deviation of a series of values placed in an array:

```
ARRAY REAL($ArrGrades;0)
QUERY([Exams];[Exams]Exam_Date=!01/07/11!)
SELECTION TO ARRAY([Exams]Exam_Grade;$ArrGrades)
vStdDev:=Std deviation($ArrGrades)
```

# Sum

Sum ( series {; attributePath} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| series | Field, Array | ➡ | Data for which to return the sum |
| attributePath | Text | ➡ | Path of attribute for which to return the sum |
| Function result | Real | ⮌ | Sum for series |

## Description

The **Sum** command returns the sum (total of all values) for *series*. If *series* is an indexed field, the index is used to total the values.

You can pass an array (one or two dimensions) in *series*. In this case, the array must be of the Integer, Longint or Real type.

This command accepts an optional *attributePath* parameter of the Text type, that you can use if *series* is an object field. It allows you to define the path of the attribute to compute. Use the standard dot notation to define paths to nested attributes, for example "company.address.number". Keep in mind that object attribute names are case-sensitive.
Only numeric attribute values are computed. If there are values in the attribute path which are not of a numeric type, they are ignored.

If the command is correctly executed, the OK system variable is set to 1. If it is interrupted (for example if the user clicks on the **Stop** button of the progress thermometer), the OK variable is set to 0.

## Example 1

The following example is an object method for a *vTotal* variable placed in a form. The object method assigns the sum of all salaries to *vTotal*:

```
vTotal:=Sum([Employees]Salary)
```

The following method is called to print the records in the selection and to activate break processing:

```
ALL RECORDS([Employees])
ORDER BY([Employees];[Employees]LastNm;>)
BREAK LEVEL(1)
ACCUMULATE([Employees]Salary)
OUTPUT FORM([Employees];"PrintForm")
PRINT SELECTION([Employees])
```

**Note:** The parameter to the **BREAK LEVEL** command should be equal to the number of breaks in your report. For more information about break processing, refer to the printing commands.

## Example 2

This example gets the sum of all the values placed in an array:

```
ARRAY REAL($ArrGrades;0)
QUERY([Exams];[Exams]Exam_Date=!01/07/11!)
SELECTION TO ARRAY([Exams]Exam_Grade;$ArrGrades)
vSum:=Sum($ArrGrades)
```

## Example 3

For an example of computing an object field attribute, please refer to the example 3 of the **Average** command description.

# Sum squares

Sum squares ( series ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| series | Field, Array | ➡ | Data for which to return the sum of squares |
| Function result | Real | ⮌ | Sum of squares of series |

## Description

**Sum squares** returns the sum of the squares of *series*. If *series* is an indexed field, the index is used to find the sum of the squares.

You can pass an array (one or two dimensions) in *series*. In this case, the array must be of the Integer, Longint or Real type.

## Example 1

The following example is an object method for the variable *vSquares*. The object method assigns the sum of squares for a data series to *vSquares*. The *vSquares* variable is printed in the last break of the report:

```
vSquares:=Sum squares([Table1]DataSeries)
```

The following method is called to print the records in the selection and to activate break processing:

```
ALL RECORDS([Table1])
ORDER BY([Table1];[Table1]DataSeries;>)
BREAK LEVEL(1)
ACCUMULATE([Table1]DataSeries)
OUTPUT FORM([Table1];"PrintForm")
PRINT SELECTION([Table1])
```

**Note:** The parameter to the **BREAK LEVEL** command should be equal to the number of breaks in your report. For more information about break processing, refer to the chapter **Printing**.

## Example 2

This example gets the sum of the squares of the values placed in an array:

```
ARRAY REAL($ArrGrades;0)
QUERY([Exams];[Exams]Exam_Date=!01/07/11!)
SELECTION TO ARRAY([Exams]Exam_Grade;$ArrGrades)
vSumSquares:=Sum squares($ArrGrades)
```

# Variance

Variance ( series ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| series | Field, Array | ⇒ | Data for which to return the variance |
| Function result | Real | ⇐ | Variance of series |

## Description

**Variance** returns the variance for *series*. If *series* is an indexed field, the index is used to find the variance.

You can pass an array (one or two dimensions) in *series*. In this case, the array must be of the Integer, Longint or Real type.

The variance of a set of values is their average squared deviation from the mean. Its measures the dispersion of values around the mean. 4D uses the following variance formula:

**Variance(x) = Sum (x-m)*(x-m)/(n-1)**
*m = Mean*
*n = Number of values*

If the values considered are not a sample, multiple the value returned by **Variance** by (n-1)/n.

## Example 1

The following example is an object method for the variable *var*. The object method assigns the sum of squares for a data series to *var*:

```
var:=Variance(Students]Grades)
```

The following method is called to print the records in the selection and to activate break processing:

```
ALL RECORDS([Students])
ORDER BY([Students];[Students]Class;>)
BREAK LEVEL(1)
ACCUMULATE([Students]Grades)
OUTPUT FORM([Students];"PrintForm")
PRINT SELECTION([Students])
```

**Note:** The parameter to the **BREAK LEVEL** command should be equal to the number of breaks in your report. For more information about break processing, refer to the chapter **Printing**.

## Example 2

This example gets the variance of the values placed in an array:

```
ARRAY REAL($ArrGrades;0)
QUERY([Exams];[Exams]Exam_Date=!01/07/11!)
SELECTION TO ARRAY([Exams]Exam_Grade;$ArrGrades)
vVariance:=Variance($ArrGrades)
```

# Operators

- Operators
- Bitwise Operators
- Comparison Operators
- Date Operators
- Logical Operators
- Numeric Operators
- Picture Operators
- String Operators
- Time Operators

# Operators

Operators are symbols used to specify operations performed between expressions. They:

- Perform calculations on numbers, dates, and times.
- Perform string operations, Boolean operations on logical expressions, and specialized operations on pictures.
- Combine simple expressions to generate new expressions.

## Precedence

The order in which an expression is evaluated is called precedence. 4D has a strict left-to-right precedence, in which algebraic order is not observed. For example:

```
3+4*5
```

returns 35, because the expression is evaluated as 3 + 4, yielding 7, which is then multiplied by 5, with the final result of 35.

To override the left-to-right precedence, you MUST use parentheses. For example:

```
3+(4*5)
```

returns 23 because the expression (4 * 5) is evaluated first, because of the parentheses. The result is 20, which is then added to 3 for the final result of 23.

Parentheses can be nested inside other sets of parentheses. Be sure that each left parenthesis has a matching right parenthesis to ensure proper evaluation of expressions. Lack of, or incorrect use of parentheses can cause unexpected results or invalid expressions. Furthermore, if you intend to compile your applications, you must have matching parentheses —the compiler detects a missing parenthesis as a syntax error.

## The Assignment Operator

You MUST distinguish the assignment operator *:=* from the other operators. Rather than combining expressions into a new one, the assignment operator copies the value of the expression to the right of the assignment operator into the variable or field to the left of the operator. For example, the following line places the value 4 (the number of characters in the word Acme) into the variable named **MyVar**. **MyVar** is then typed as a numeric value.

```
MyVar := Length ("Acme")
```

**Important:** Do NOT confuse the assignment operator *:=* with the equality comparison operator =.

The other operators provided by the 4D language are described in the following sections:

## String Operators

See the **String Operators** section.

## Numeric Operators

See the **Numeric Operators** section.

## Date Operators

See the **Date Operators** section.

## Time Operators

See the **Time Operators** section.

## Comparison Operators

See the **Comparison Operators** section.

## Logical Operators

See the **Logical Operators** section.

## Picture Operators

See the **Picture Operators** section.

## Bitwise Operators

See the **Bitwise Operators** section.

## Bitwise Operators

---

The bitwise operators operates on **Long Integer** expressions or values.

**Note:** If you pass an *Integer* or a *Real* value to a bitwise operator, 4D evaluates the value as a **Long Integer** value before calculating the expression that uses the bitwise operator.

While using the bitwise operators, you must think about a **Long Integer** value as an array of 32 bits. The bits are numbered from 0 to 31, from right to left.

Because each bit can equal 0 or 1, you can also think about a **Long Integer** value as a value where you can store 32 **Boolean** values. A bit equal to 1 means True and a bit equal to *0* means False.

An expression that uses a bitwise operator returns a **Long Integer** value, except for the **Bit Test** operator, where the expression returns a **Boolean** value. The following table lists the bitwise operators and their syntax:

| Operation | Operator | Syntax | Returns | |
|---|---|---|---|---|
| Bitwise AND | & | Long & Long | Long | |
| Bitwise OR (inclusive) | \| | Long \| Long | Long | |
| Bitwise OR (exclusive) | ^\| | Long ^\| Long | Long | |
| Left Bit Shift | << | Long << Long | Long | (see note 1) |
| Right Bit Shift | >> | Long >> Long | Long | (see note 1) |
| Bit Set | ?+ | Long ?+ Long | Long | (see note 2) |
| Bit Clear | ?- | Long ?- Long | Long | (see note 2) |
| Bit Test | ?? | Long ?? Long | Boolean | (see note 2) |

**Notes**

1. For the **Left Bit Shift** and **Right Bit Shift** operations, the second operand indicates the number of positions by which the bits of the first operand will be shifted in the resulting value. Therefore, this second operand should be between *0* and *31*. Note however, that shifting by *0* returns an unchanged value and shifting by more than *31* bits returns 0x00000000 because all the bits are lost. If you pass another value as second operand, the result is non-significant.
2. For the **Bit Set**, **Bit Clear** and **Bit Test** operations , the second operand indicates the number of the bit on which to act. Therefore, this second operand must be between *0* and *31*; otherwise, the result of the expression is non-significant.

The following table lists the bitwise operators and their effects:

| Operation | Description |
|---|---|
| Bitwise AND | Each resulting bit is the logical AND of the bits in the two operands.<br>Here is the logical AND table:<br>*1 & 1 --> 1*<br>*0 & 1 --> 0*<br>*1 & 0 --> 0*<br>*0 & 0 --> 0*<br>In other words, the resulting bit is *1* if the two operand bits are *1*;otherwise the resulting bit is *0*. |
| Bitwise OR (inclusive) | Each resulting bit is the logical OR of the bits in the two operands.<br>Here is the logical OR table:<br>*1 \| 1 --> 1*<br>*0 \| 1 --> 1*<br>*1 \| 0 --> 1*<br>*0 \| 0 --> 0*<br>In other words, the resulting bit is *1* if at least one of the two operand bits is *1*; otherwise the resulting bit is *0*. |
| Bitwise OR (exclusive) | Each resulting bit is the logical XOR of the bits in the two operands.<br>Here is the logical XOR table:<br>*1 ^\| 1 --> 0*<br>*0 ^\| 1 --> 1*<br>*1 ^\| 0 --> 1*<br>*0 ^\| 0 --> 0*<br>In other words, the resulting bit is *1* if only one of the two operand bits is *1*; otherwise the resulting bit is *0*. |
| Left Bit Shift | The resulting value is set to the first operand value, then the resulting bits are shifted to the left by the number of positions indicated by the second operand. The bits on the left are lost and the new bits on the right are set to 0.<br>**Note:** Taking into account only positive values, shifting to the left by N bits is the same as multiplying by *2^N*. |
| Right Bit Shift | The resulting value is set to the first operand value, then the resulting bits are shifted to the right by the number of position indicated by the second operand. The bits on the right are lost and the new bits on the left are set to 0.<br>**Note:** Taking into account only positive values, shifting to the right by N bits is the same as dividing by *2^N*. |
| Bit Set | The resulting value is set to the first operand value, then the resulting bit, whose number is indicated by the second operand, is set to *1*. The other bits are left unchanged. |
| Bit Clear | The resulting value is set to the first operand value, then the resulting bit, whose number is indicated by the second operand, is set to *0*. The other bits are left unchanged. |
| Bit Test | Returns True if, in the first operand, the bit whose number is indicated by the second operand is equal to *1.* Returns False if, in the first operand, the bit whose number is indicated by the second operand is equal to *0*. |

## Example

1) The following table gives an example of each bit operator:

| Operation | Example | Result |
|---|---|---|
| Bitwise AND | 0x0000FFFF & 0xFF00FF00 | 0x0000FF00 |
| Bitwise OR (inclusive) | 0x0000FFFF \| 0xFF00FF00 | 0xFF00FFFF |
| Bitwise OR (exclusive) | 0x0000FFFF ^\| 0xFF00FF00 | 0xFF0000FF |
| Left Bit Shift | 0x0000FFFF << 8 | 0x00FFFF00 |
| Right Bit Shift | 0x0000FFFF >> 8 | 0x000000FF |
| Bit Set | 0x00000000 ?+ 16 | 0x00010000 |
| Bit Clear | 0x00010000 ?- 16 | 0x00000000 |
| Bit Test | 0x00010000 ?? 16 | True |

2) 4D provides many predefined constants. The literals of some of these constants end with “bit” or “mask.” For example, this is the case of the constants provided in the **Resources Properties** theme:

| Constant | Type | Value |
|---|---|---|
| Changed resource bit | Longint | 1 |
| Changed resource mask | Longint | 2 |
| Locked resource bit | Longint | 4 |
| Locked resource mask | Longint | 16 |
| Preloaded resource bit | Longint | 2 |
| Preloaded resource mask | Longint | 4 |
| Protected resource bit | Longint | 3 |
| Protected resource mask | Longint | 8 |
| Purgeable resource bit | Longint | 5 |
| Purgeable resource mask | Longint | 32 |
| System heap resource bit | Longint | 6 |
| System heap resource mask | Longint | 64 |

These constants enable you to test the value returned by **Get resource properties** or to create the value passed to **_o_SET RESOURCE PROPERTIES**. Constants whose literal ends with “bit” give the position of the bit you want to test, clear, or set. Constants whose literal ends with “mask” gives a long integer value where only the bit (that you want to test, clear, or set) is equal to one.

For example, to test whether a resource (whose properties have been obtained in the variable *$vlResAttr*) is purgeable or not, you can write:

```
If($vlResAttr ?? Purgeable resource bit) ` Is the resource purgeable?
```

or:

```
If(($vlResAttr  & Purgeable resource mask)#0)Is the resource purgeable?
```

Conversely, you can use these constants to set the same bit. You can write:

```
$vlResAttr:=$vlResAttr ?+Purgeable resource bit
```

or:

```
$vlResAttr:=$vlResAttr |Purgeable resource bit
```

3) This example stores two *Integer* values into a *Long Integer* value. You can write:

```
$vlLong:=($viIntA<<16)|$viIntB  ` Store two Integers in a Long Integer
$vlIntA:=$vlLong>>16 ` Extract back the integer stored in the high-word
$viIntB:=$vlLong  & 0xFFFF ` Extract back the Integer stored in the low-word
```

**Tip:** Be careful when manipulating *Long Integer* or *Integer* values with expressions that combine numeric and bitwise operators. The high bit (bit 31 for Long Integer, bit 15 for Integer) sets the sign of the value — positive if it is cleared, negative if it is set. Numeric operators use this bit for detecting the sign of a value, bitwise operators do not care about the meaning of this bit.

## Comparison Operators

The tables in this section show the comparison operators as they apply to string, numeric, date, time, pointer and picture with metadata expressions (you cannot use them with array or BLOB type expressions). An expression that uses a comparison operator returns a Boolean value, either **TRUE** or **FALSE**.

**Note:** You can compare two pictures using the **Equal pictures** command.

### String Comparisons

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| Equality | String = String | Boolean | "abc" = "abc" | True |
| | | | "abc" = "abd" | False |
| Inequality | String # String | Boolean | "abc" # "abd" | True |
| | | | "abc" # "abc" | False |
| Greater than | String > String | Boolean | "abd" > "abc" | True |
| | | | "abc" > "abc" | False |
| Less than | String < String | Boolean | "abc" < "abd" | True |
| | | | "abc" < "abc" | False |
| Greater than or equal to | String >= String | Boolean | "abd" >= "abc" | True |
| | | | "abc" >= "abd" | False |
| Less than or equal to | String <= String | Boolean | "abc" <= "abd" | True |
| | | | "abd" <= "abc" | False |
| Contains keyword | String % String | Boolean | "Alpha Bravo" % "Bravo" | True |
| | | | "Alpha Bravo" % "ravo" | False |
| | Picture % String | Boolean | *Picture_expr* % "Mer" | True (*) |

**(*)** If the keyword "Mer" is associated with the picture stored in the picture expression (field or variable).
**Important:** Additional information about string comparisons are provided at the end of this section.

### Numeric Comparisons

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| Equality | Number = Number | Boolean | 10 = 10 | True |
| | | | 10 = 11 | False |
| Inequality | Number # Number | Boolean | 10 #11 | True |
| | | | 10 # 10 | False |
| Greater than | Number > Number | Boolean | 11 > 10 | True |
| | | | 10 > 11 | False |
| Less than | Number < Number | Boolean | 10 < 11 | True |
| | | | 11 < 10 | False |
| Greater than or equal to | Number >= Number | Boolean | 11 >= 10 | True |
| | | | 10 >= 11 | False |
| Less than or equal to | Number <= Number | Boolean | 10 <= 11 | True |
| | | | 11 <= 10 | False |

### Date Comparisons

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| Equality | Date = Date | Boolean | !1/1/97! =!1/1/97! | True |
| | | | !1/20/97! =!1/1/97! | False |
| Inequality | Date # Date | Boolean | !1/20/97! # !1/1/97! | True |
| | | | !1/1/97! # !1/1/97! | False |
| Greater than | Date > Date | Boolean | !1/20/97! > !1/1/97! | True |
| | | | !1/1/97! > !1/1/97! | False |
| Less than | Date < Date | Boolean | !1/1/97! < !1/20/97! | True |
| | | | !1/1/97! < !1/1/97! | False |
| Greater than or equal to | Date >= Date | Boolean | !1/20/97! >=!1/1/97! | True |
| | | | !1/1/97!>=!1/20/97! | False |
| Less than or equal to | Date <= Date | Boolean | !1/1/97!<=!1/20/97! | True |
| | | | !1/20/97!<=!1/1/97! | False |

## Time Comparisons

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| Equality | Time = Time | Boolean | ?01:02:03? = ?01:02:03? | True |
| | | | ?01:02:03? = ?01:02:04? | False |
| Inequality | Time # Time | Boolean | ?01:02:03? # ?01:02:04? | True |
| | | | ?01:02:03? # ?01:02:03? | False |
| Greater than | Time > Time | Boolean | ?01:02:04? > ?01:02:03? | True |
| | | | ?01:02:03? > ?01:02:03? | False |
| Less than | Time < Time | Boolean | ?01:02:03? < ?01:02:04? | True |
| | | | ?01:02:03? < ?01:02:03? | False |
| Greater than or equal to | Time >= Time | Boolean | ?01:02:03? >=?01:02:03? | True |
| | | | ?01:02:03? >=?01:02:04? | False |
| Less than or equal to | Time <= Time | Boolean | ?01:02:03? <=?01:02:03? | True |
| | | | ?01:02:04? <=?01:02:03? | False |

## Pointer comparisons

With:

```
  ` vPtrA and vPtrB point to the same object
vPtrA:=->anObject
vPtrB:=->anObject
  ` vPtrC points to another object
vPtrC:=->anotherObject
```

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| Equality | Pointer = Pointer | Boolean | vPtrA = vPtrB | True |
| | | | vPtrA = vPtrC | False |
| Inequality | Pointer # Pointer | Boolean | vPtrA # vPtrC | True |
| | | | vPtrA # vPtrB | False |

## More about string comparisons

- Strings are compared on a character-by-character basis (except in the case of searching by keywords, see below).
- When strings are compared, the case of the characters is ignored; thus, "a"="A" returns TRUE. To test if the case of two characters is different, compare their character codes. For example, the following expression returns FALSE:

```
Character code("A")=Character code("a") // because 65 is not equal to 97
```

- When strings are compared, diacritical characters are compared using the system character comparison table of your computer. For example, the following expressions return TRUE:

```
"n"="ñ"
"n"="Ñ"
"A"="å"
 // and so on
```

- Unlike other string comparisons, searching by keywords looks for “words” in “texts”: words are considered both individually and as a whole. The % operator always returns False if the query concerns several words or only part of a word (for example, a syllable). The “words” are character strings surrounded by “separators,” which are spaces and punctuation characters and dashes. An apostrophe, like in “Today's”, is usually considered as part of the word, but will be ignored in certain cases (see the rules below). Numbers can be searched for because they are evaluated as a whole (including decimal symbols). Other symbols (currency, temperature, and so on) will be ignored.

```
"Alpha Bravo Charlie"%"Bravo" // Returns True
"Alpha Bravo Charlie"%"vo" ` Returns False
"Alpha Bravo Charlie"%"Alpha Bravo" // Returns False
"Alpha,Bravo,Charlie"%"Alpha" // Returns True
"Software and Computers"%"comput@" // Returns True
```

**Notes**:
- 4D uses the ICU library for detecting keywords. For more information about the rules implemented, please refer to the following address:http://www.unicode.org/unicode/reports/tr29/#Word_Boundaries.
- In the Japanese version, instead of ICU, 4D uses *Mecab* by default for detecting keywords. For more information, please refer to **Support of Mecab (Japanese version)**.

- The wildcard character (@) can be used in any string comparison to match any number of characters. For example, the following expression is TRUE:

```
"abcdefghij"="abc@"
```

The wildcard character must be used within the second operand (the string on the right side) in order to match any number of characters. The following expression is FALSE, because the @ is considered only as a one character in the first operand:

```
"abc@"="abcdefghij"
```

The wildcard means “one or more characters or nothing”. The following expressions are TRUE:

```
"abcdefghij"="abcdefghij@"
"abcdefghij"="@abcdefghij"
"abcdefghij"="abcd@efghij"
"abcdefghij"="@abcdefghij@"
"abcdefghij"="@abcde@fghij@"
```

On the other hand, whatever the case, a string comparison with two consecutive wildcards will always return FALSE. The following expression is FALSE:

```
"abcdefghij"="abc@@fg"
```

When the comparison operator is or contains a < or > symbol, only comparison with a single wildcard located at the end of the operand is supported:

```
"abcd"<="abc@" // Valid comparison
"abcd"<="abc@ef" //Not a valid comparison
```

**Tip:**If you want to execute comparisons or queries using @ as a character (and not as a wildcard), you have two options:

- Use the **Character code (**At sign**)** instruction.
Imagine, for example, that you want to know if a string ends with the @ character.
- the following expression (if *$vsValue* is not empty) is always TRUE:

```
($vsValue≤Length($vsValue)≥="@")
```

- the following expression will be evaluated correctly:

```
(Character code($vsValue≤Length($vsValue)≥)#64)
```

- Use the "Consider @ as a wildcard only when at the beginning or end of text patterns" option which can be accessed using the Database Settings dialog box.
  This option lets you define how the @ character is interpreted when it is included in a character string. As such, it can influence how comparison operators are used in Query or Order By. For more information, refer to the 4D *Design Reference* manual.

## Date Operators

---

An expression that uses a date operator returns a date or a number, depending on the operation. All date operations will result in an accurate date, taking into account the change between years and leap years. The following table shows the date operators:

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| Date difference | Date – Date | Number | !1997-01-20! - !1997-01-01! | 19 |
| Day addition | Date + Number | Date | !1997-01-20! + 9 | !1997-01-29! |
| Day subtraction | Date – Number | Date | !1997-01-20! - 9 | !1997-01-11! |

## Logical Operators

---

4D supports two logical operators that work on Boolean expressions: conjunction (AND) and inclusive disjunction (OR). A logical AND returns TRUE if both expressions are TRUE. A logical OR returns TRUE if at least one of the expressions is TRUE.

4D also provides the Boolean functions *True*, *False*, and *Not*. For more information, see the descriptions of these commands.

The following table shows the logical operators:

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| AND | Boolean & Boolean | Boolean | ("A" = "A") & (15 # 3) | True |
| | | | ("A" = "B") & (15 # 3) | False |
| | | | ("A" = "B") & (15 = 3) | False |
| OR | Boolean \| Boolean | Boolean | ("A" = "A") \| (15 # 3) | True |
| | | | ("A" = "B") \| (15 # 3) | True |
| | | | ("A" = "B") \| (15 = 3) | False |

The following is the truth table for the AND logical operator:

| **Expr1** | **Expr2** | **Expr1** & **Expr2** |
|---|---|---|
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

The following is the truth table for the OR logical operator:

| **Expr1** | **Expr2** | **Expr1 \| Expr2** |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

**Tip**: If you need to calculate the exclusive disjunction between Expr1 and Expr2, evaluate:

```
(Expr1|Expr2) & Not(Expr1 & Expr2)
```

## Numeric Operators

---

An expression that uses a numeric operator returns a number. The following table shows the numeric operators:

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| Addition | Number + Number | Number | 2 + 3 | 5 |
| Subtraction | Number – Number | Number | 3 – 2 | 1 |
| Multiplication | Number * Number | Number | 5 * 2 | 10 |
| Division | Number /Number | Number | 5 / 2 | 2.5 |
| Longint division | Number ¥ Number | Number | 5 ¥ 2 | 2 |
| Modulo | Number % Number | Number | 5 % 2 | 1 |
| Exponentiation | Number ^ Number | Number | 2 ^ 3 | 8 |

The modulo operator % divides the first number by the second number and returns a whole number remainder. Here are some examples:

- 10 % 2 returns 0 because 10 is evenly divided by 2.
- 10 % 3 returns 1 because the remainder is 1.
- 10.5 % 2 returns 0 because the remainder is not a whole number.

**WARNING**:

- The modulo operator % returns significant values with numbers that are in the Long Integer range (from minus 2^31 to 2^31 minus one). To calculate the modulo with numbers outside of this range, use the **Mod** command.
- The longint division operator ¥ returns significant values with integer numbers only.

## Picture Operators

An expression that uses a picture operator returns a picture. The following table shows the picture operators.

| Operation | Syntax | Action |
|---|---|---|
| Horizontal concatenation | Pict1 + Pict2 | Add Pict2 to the right of Pict1 |
| Vertical concatenation | Pict1 / Pict2 | Add Pict2 to the bottom of Pict1 |
| Exclusive superimposition(*) | Pict1 & Pict2 | Superimposes Pict2 on top of Pict1 (Pict2 in foreground) |
| Inclusive superimposition(*) | Pict1 \| Pict2 | Superimposes Pict2 on Pict1 and returns resulting mask if both pictures are the same size |
| Horizontal move | Picture + Number | Move Picture horizontally Number pixels |
| Vertical move | Picture / Number | Move Picture vertically Number pixels |
| Resizing | Picture * Number | Resize Picture by Number ratio |
| Horizontal scaling | Picture *+ Number | Resize Picture horizontally by Number ratio |
| Vertical scaling | Picture */ Number | Resize Picture vertically by Number ratio |

**(*)** The functioning of the exclusive superimposition (&) and inclusive superimposition (|) operators is modified starting with 4Dv14 following the update of the display management libraries used by the program.
**Pict3 := Pict1 & Pict2** produces the same result as:

```
COMBINE PICTURES(pict3;pict1;Superimposition;pict2)
```

**Pict3 := Pict1 | Pict2** produces the same result as:

```
$equal:=Equal pictures(Pict1;Pict2;Pict3)
```

Note that in order to use the **|** operator, Pict1 and Pict2 must have exactly the same dimension. If both pictures are a different size, the operation Pict1 | Pict2 produces a blank picture.

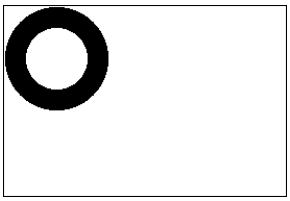
**Note:** The **COMBINE PICTURES** command can be used to superimpose pictures while keeping the characteristics of each source picture in the resulting picture.

The picture operators return vectorial pictures if the two source pictures are vectorial. Remember, however, that pictures printed by the display format On Background are printed bitmapped.

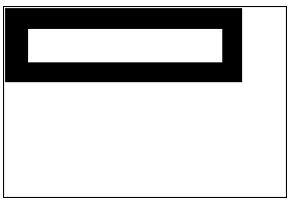
## Example

In the following examples, all of the pictures are shown using the display format On Background.

Here is the picture *circle*:



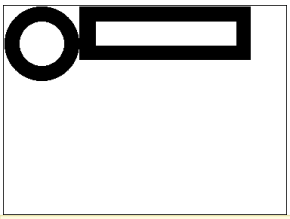
Here is the picture *rectangle*:



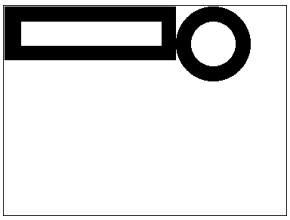
In the following examples, each expression is followed by its graphical representation.

- Horizontal concatenation

```
circle+rectangle ` Place the rectangle to the right of the circle
```

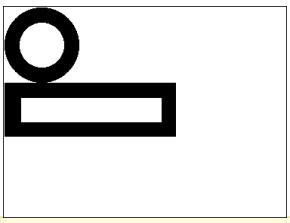


```
rectangle+circle ` Place the circle to the right of the rectangle
```

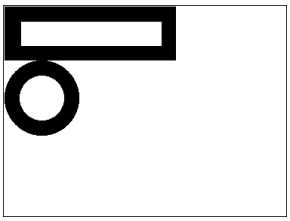


- Vertical concatenation

```
circle/rectangle ` Place the rectangle under the circle
```

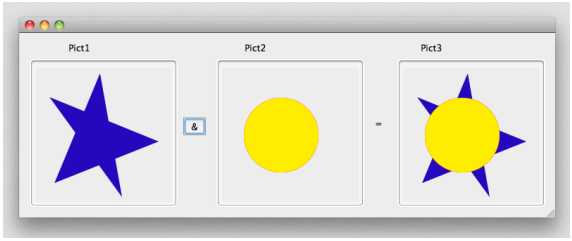


```
rectangle/circle ` Place the circle under the rectangle
```



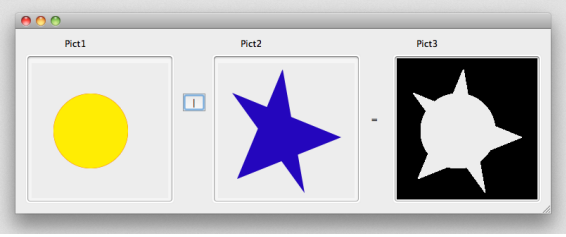
- Exclusive superimposition

```
Pict3:=Pict1 & Pict2 // Superimposes Pict2 on top of  Pict1
```



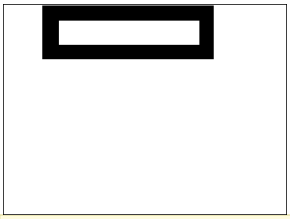

- Inclusive superimposition

```
Pict3:=Pict1|Pict2 // Recovers resulting mask from superimposing two pictures of the same size
```

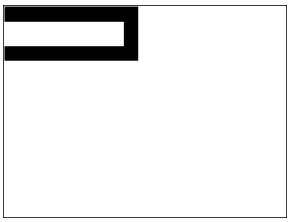



- Horizontal move

```
rectangle+50 ` Move the rectangle 50 pixels to the right
```

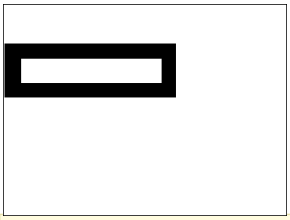


```
rectangle-50 ` Move the rectangle 50 pixels to the left
```

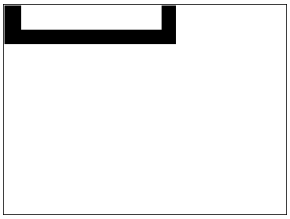


- Vertical move

```
rectangle/50 ` Move the rectangle down by 50 pixels
```

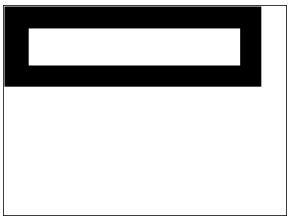


```
rectangle/-20 ` Move the rectangle up by 20 pixels
```

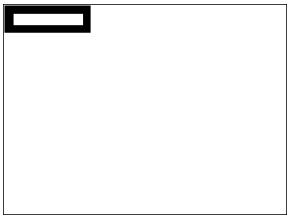


- Resize

```
rectangle*1.5 ` The rectangle becomes 50% bigger
```

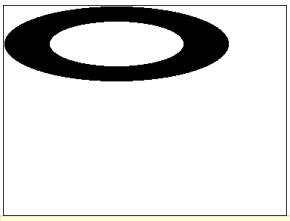


```
rectangle*0.5 ` The rectangle becomes 50% smaller
```

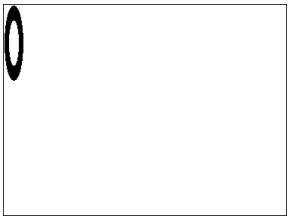


- Horizontal scaling

```
circle*+3 ` The circle becomes 3 times wider
```

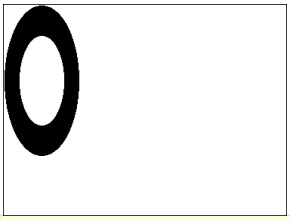


```
circle*+0.25 ` The circle's width becomes a quarter of what it was
```

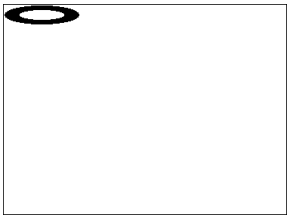


- Vertical scaling

```
circle*/2 ` The circle becomes twice as tall
```



```
circle*/0.25 ` The circle's height becomes a quarter of what it was
```



## String Operators

---

An expression that uses a string operator returns a string. The following table shows the string operators:

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| Concatenation | String + String | String | "abc" + "def" | "abcdef" |
| Repetition | String * Number | String | "ab" * 3 | "ababab" |

## Time Operators

An expression that uses a time operator returns a time or a number, depending on the operation. The following table shows the time operators:

| Operation | Syntax | Returns | Expression | Value |
|---|---|---|---|---|
| Addition | Time + Time | Time | ?02:03:04? + ?01:02:03? | ?03:05:07? |
| Subtraction | Time – Time | Time | ?02:03:04? – ?01:02:03? | ?01:01:01? |
| Addition | Time + Number | Number | ?02:03:04? + 65 | 7449 |
| Subtraction | Time – Number | Number | ?02:03:04? – 65 | 7319 |
| Multiplication | Time * Number | Number | ?02:03:04? * 2 | 14768 |
| Division | Time / Number | Number | ?02:03:04? / 2 | 3692 |
| Longint division | Time ¥ Number | Number | ?02:03:04? ¥ 2 | 3692 |
| Modulo | Time % Time | Time | ?20:10:00? % ?04:20:00? | ?02:50:00? |
| Modulo | Time % Number | Number | ?02:03:04? % 2 | 0 |

### Example 1

To obtain a time expression from an expression that combines a time expression with a number, use the commands **Time** and **Time string**.

You can combine expressions of the time and number types using the **Time** or **Current time** functions. For example:

Example:

```
 //The following line assigns to $vlSeconds the number of seconds that will be elapsed between midnight and one hour from now
$vlSeconds:=Current time+3600
 //The following line assigns to $vHSoon the time it will be in one hour
$vhSoon:=Time(Current time+3600)
```

The second line could be written in a simpler way:

```
 // The following line assigns to $vHSoon the time it will be in one hour
$vhSoon:=Current time+?01:00:00?
```

### Example 2

Some situations may require you to convert a time expression into a numeric expression.
For example, you open a document using **Open document**, which returns a Document Reference (*DocRef*) that is formally a time expression. Later, you want to pass that *DocRef* to a 4D Extension routine that expects a numeric value as document reference. In such a case, use the addition with 0 (zero) to get a numeric value from the time value, but without changing its value.

Example:

```
  ` Select and open a document
$vhDocRef:=Open document("")
If(OK=1)
  ` Pass the DocRef time expression as a numeric expresssion to a 4D Extension routine
   DO SOMETHING SPECIAL(0+$vhDocRef)
End if
```

### Example 3

The Modulo operator can be used, more specifically, to add times that take the 24-hour format into account:

```
$t1:=?23:00:00? // It is 23:00 p.m.
 // We want to add 2 and a half hours
$t2:=$t1 +?02:30:00? // With a simple addition, $t2 is ?25:30:00?
$t2:=($t1 +?02:30:00?)%?24:00:00? // $t2 is ?01:30:00? and it is 1:30 a.m. the next morning
```

# Pasteboard

- Managing Pasteboards
- APPEND DATA TO PASTEBOARD
- CLEAR PASTEBOARD
- Get file from pasteboard
- GET PASTEBOARD DATA
- GET PASTEBOARD DATA TYPE
- GET PICTURE FROM PASTEBOARD
- Get text from pasteboard
- Pasteboard data size
- SET FILE TO PASTEBOARD
- SET PICTURE TO PASTEBOARD
- SET TEXT TO PASTEBOARD

# Managing Pasteboards

The commands of the “Pasteboard” theme can be used both for managing copy/paste actions (Clipboard management), as well as inter-application drag and drop actions.

4D uses two data pasteboards: one for copied (or cut) data, which is the actual clipboard that was already present in previous versions, and the other for data being dragged and dropped.
These two pasteboards are managed using the same commands. You access one or the other depending on the context:

- The drag and drop pasteboard can only be accessed within the On Begin Drag Over, On Drag over or On Drop form events and in the **On Drop database method**. Outside of these contexts, the drag and drop pasteboard is not available.
- The copy/paste pasteboard can be accessed in all other cases. Unlike the drag and drop pasteboard, it keeps the data that are placed in it during the entire session. so long as they are not cleared or reused.

## Types of Data

During drag and drop actions, different types of data can be placed on and read from the pasteboard. You can access a data type in several ways:

- Via its 4D signature: The 4D signature is a character string indicating a data type referenced by the 4D application. The use of 4D signatures facilitates the development of multi-platform applications since these signatures are identical under Mac OS and Windows. You will find the list of 4D signatures below.
- Via a UTI (Uniform Type Identifier, Mac OS only): The UTI standard, specified by Apple, associates a character string with each type of native object. For example, GIF pictures have the UTI type “com.apple.gif”. UTI types are published in Apple documentations as well as by the editors concerned.
- Via its number or its format name (Windows only): Under Windows, each native data type is referenced by its number (“3”, “12”, and so on) and a name (“Rich Text Edit”). By default, Microsoft specifies several native types called standard data formats. In addition, third-party editors can “save” format names in the system, which then attributes them a number in return. For more information about this and about native types, please refer to the Microsoft developer documentation (more particularly at http://msdn2.microsoft.com/en-us/library/ms649013.aspx).

**Note:** In 4D commands, the Windows format numbers are handled as text.

All the commands of the “Pasteboard” theme can work with each one of these data types. You can find out which data types are present in the pasteboard in each of these formats using the **GET PASTEBOARD DATA TYPE** command.

**Note:** The 4-character types (TEXT, PICT or custom types) are kept for compatibility with prior versions of 4D.

## 4D Signatures

Here is the list of standard 4D signatures as well as their description:

| Signature | Description |
|---|---|
| "com.4d.private.text.native" | Text in native character set |
| "com.4d.private.text.utf16" | Text in Unicode character set |
| "com.4d.private.text.rtf" | Enriched text |
| "com.4d.private.picture.pict" | PICT picture format |
| "com.4d.private.picture.png" | PNG picture format |
| "com.4d.private.picture.gif" | GIF picture format |
| "com.4d.private.picture.jfif" | JPEG picture format |
| "com.4d.private.picture.emf" | EMF picture format |
| "com.4d.private.picture.bitmap" | BITMAP picture format |
| "com.4d.private.picture.tiff" | TIFF picture format |
| "com.4d.private.picture.pdf" | PDF document |
| "com.4d.private.file.url" | File pathname |

# APPEND DATA TO PASTEBOARD

APPEND DATA TO PASTEBOARD ( dataType ; data )

| Parameter | Type | | Description |
|---|---|---|---|
| dataType | String | ⇒ | Type of data to be added |
| data | BLOB | ⇒ | Data to append to the pasteboard |

## Description

The **APPEND DATA TO PASTEBOARD** command appends to the pasteboard the data contained in the BLOB *data* under the data type specified in *dataType*.

**Note:** In the case of copy/paste operations, the pasteboard is equivalent to the Clipboard.

In *dataType*, pass a value specifying the type of data to be added. You can pass a 4D signature, a UTI type (Mac OS), a format name/number (Windows), or a 4-character type (compatibility). For more information about these types, please refer to the **Managing Pasteboards** section.

**Note for Windows users:** When the command is used with Text type data (*dataType* is "TEXT", com.4d.private.text.native or com.4d.private.text.utf16), the string contained in the BLOB parameter *data* must end with the NULL character under Windows.

Usually, you will use the **APPEND DATA TO PASTEBOARD** command to append multiple instances of the same data to the pasteboard or to append data that is not text or a picture. To append new data to the pasteboard, you must first clear the pasteboard using the **CLEAR PASTEBOARD** command.

If you want to clear and append:

- text to the pasteboard, use the **SET TEXT TO PASTEBOARD** command,
- a picture to the pasteboard, use the **SET PICTURE TO PASTEBOARD** command.
- a file pathname (drag and drop), use the **SET FILE TO PASTEBOARD** command.

However, note that if a BLOB actually contains some text or a picture, you can use the **APPEND DATA TO PASTEBOARD** command to append a text or a picture to the pasteboard.

## Example

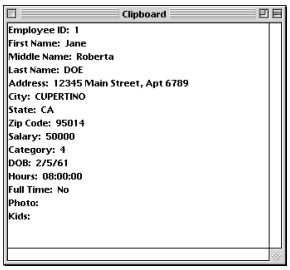
Using Pasteboard commands and BLOBs, you can build sophisticated Cut/Copy/Paste schemes that deal with structured data rather than a unique piece of data. In the following example, the two project methods ***SET RECORD TO PASTEBOARD*** and ***GET RECORD FROM PASTEBOARD*** enable you to treat a whole record as one piece of data to be copied to or from the pasteboard.

```
 // SET RECORD TO PASTEBOARD project method
 // SET RECORD TO PASTEBOARD ( Number )
 // SET RECORD TO PASTEBOARD ( Table number )

C_LONGINT($1;$vlField;$vlFieldType)
C_POINTER($vpTable;$vpField)
C_STRING(255;$vsDocName)
C_TEXT($vtRecordData;$vtFieldData)
C_BLOB($vxRecordData)

 // Clear the pasteboard (it will stay empty if there is no current record)
CLEAR PASTEBOARD
 // Get a pointer to the table whose number is passed as parameter
$vpTable:=Table($1)
 // If there is a current record for that table
If((Record number($vpTable->)>=0)|(Is new record($vpTable->)))
 //Initialize the text variable that will hold the text image of the record
   $vtRecordData:=""
 // For each field of the record:
   For($vlField;1;Get last field number($1))
 //Get the type of the field
      GET FIELD PROPERTIES($1;$vlField;$vlFieldType)
```

```
    // Get a pointer to the field
        $vpField:=Field($1;$vlField)
    // Depending on the type of the field, copy (or not) its data in the appropriate manner
        Case of
           :(($vlFieldType=Is alpha field)|($vlFieldType=Is text))
              $vtFieldData:=$vpField->
           :(($vlFieldType=Is real)|($vlFieldType=Is integer)|($vlFieldType=Is longint)|($vlFieldType=Is date)|($vlFieldType=Is
time))
              $vtFieldData:=String($vpField->)
           :($vlFieldType=Is Boolean)
              $vtFieldData:=String(Num($vpField->);"Yes;;No")
           Else
    // Skip and ignore other field data types
              $vtFieldData:=""
        End case
    // Accumulate the field data into the text variable holding the text image of the record
        $vtRecordData:=$vtRecordData+Field name($1;$vlField)+":"+Char(9)+$vtFieldData+CR
    // Note: The method CR returns Char(13) on Macintosh and Char(13)+Char(10) on Windows
      End for
    // Put the text image of the record into the pasteboard
      SET TEXT TO PASTEBOARD($vtRecordData)
    // Name for scrap file in Temporary folder
      $vsDocName:=Temporary folder+"Scrap"+String(1+(Random%99))
    // Delete the scrap file if it exists (error should be tested here)
      DELETE DOCUMENT($vsDocName)
    // Create scrap file
      SET CHANNEL(10;$vsDocName)
    // Send the whole record into the scrap file
      SEND RECORD($vpTable->)
    // Close the scrap file
      SET CHANNEL(11)
    // Load the scrap file into a BLOB
      DOCUMENT TO BLOB($vsDocName;$vxRecordData)
    // We longer need the scrap file
      DELETE DOCUMENT($vsDocName)
    // Append the full image of the record into the pasteboard
    // Note: We use arbitrarily "4Drc" as data type
      APPEND DATA TO PASTEBOARD("4Drc";$vxRecordData)
    // At this point, the pasteboardcontains:
    // (1) A text image of the record (as shown in the screen shots below)
    // (2) A whole image of the record (Picture, Subfile and BLOB fields included)
   End if
```

While entering the following record:


If you apply the method ***SET RECORD TO PASTEBOARD*** to the [Employees] table, the pasteboard will contain the text image of the record, as shown, and also the whole image of the record.



You can paste this image of the record to another record, using the method ***GET RECORD FROM PASTEBOARD***, as follows:

```
 // GET RECORD FROM PASTEBOARD method
 // GET RECORD FROM PASTEBOARD( Number )
 // GET RECORD FROM PASTEBOARD( Table number )
C_LONGINT($1;$vlField;$vlFieldType;$vlPosCR;$vlPosColon)
C_POINTER($vpTable;$vpField)
C_STRING(255;$vsDocName)
C_BLOB($vxPasteboardData)
C_TEXT($vtPasteboardData;$vtFieldData)

 // Get a pointer to the table whose number is passed as parameter
$vpTable:=Table($1)
 // If there is a current record
If((Record number($vpTable->)>=0)|(Is new record($vpTable->)))
   Case of
 // Does the pasteboard contain a full image record?
      :(Pasteboard data size("4Drc")>0)
 //  If so, extract the pasteboard contents
         GET PASTEBOARD DATA("4Drc";$vxPasteboardData)
 // Name for scrap file in Temporary folder
         $vsDocName:=Temporary folder+"Scrap"+String(1+(Random%99))  
 // Delete the scrap file if it exists (error should be tested here)
         DELETE DOCUMENT($vsDocName)
 // Save the BLOB into the scrap file
         BLOB TO DOCUMENT($vsDocName;$vxPasteboardData)
 // Open the scrap file
         SET CHANNEL(10;$vsDocName)
 // Receive the whole record from the scrap file
         RECEIVE RECORD($vpTable->)
 // Close the scrap file
         SET CHANNEL(11)
 // We longer need the scrap file
         DELETE DOCUMENT($vsDocName)
 // Does the pasteboard contain TEXT?
      :(Pasteboard data size("TEXT")>0)
 // Extract the text from the pasteboard
         $vtPasteboardData:=Get text from pasteboard
 // Initialize field number to be increment
         $vlField:=0
         Repeat
 // Look for the next field line in the text
            $vlPosCR:=Position(CR;$vtPasteboardData)
            If($vlPosCR>0)
 // Extract the field line
               $vtFieldData:=Substring($vtPasteboardData;1;$vlPosCR-1)
 // If there is a colon ":"
               $vlPosColon:=Position(":";$vtFieldData)
               If($vlPosColon>0)
 // Take only the field data (eliminate field name)
                  $vtFieldData:=Substring($vtFieldData;$vlPosColon+2)
               End if
 // Increment field number
               $vlField:=$vlField+1
 // Pasteboard may contain more data than we need...
               If($vlField<=Get last field number($vpTable))
 // Get the type of the field
                  GET FIELD PROPERTIES($1;$vlField;$vlFieldType)
 // Get a pointer to the field
                  $vpField:=Field($1;$vlField)
 // Depending on the type of the field, copy (or not) the text in the appropriate manner
                  Case of
                     :(($vlFieldType=Is alpha field)|($vlFieldType=Is text))
                        $vpField->:=$vtFieldData
                     :(($vlFieldType=Is real)|($vlFieldType=Is integer)|($vlFieldType=Is longint))
                        $vpField->:=Num($vtFieldData)
                     :($vlFieldType=Is date)
                        $vpField->:=Date($vtFieldData)
                     :($vlFieldType=Is time)
                        $vpField->:=Time($vtFieldData)
```

```
                    :($vlFieldType=Is Boolean)
                       $vpField->:=($vtFieldData="Yes")
                    Else
   // Skip and ignore other field data types
                 End case
              Else
   // All fields have been assigned, get out of the loop
                 $vtPasteboardData:=""
              End if
   // Eliminate text that has just been extracted
              $vtPasteboardData:=Substring($vtPasteboardData;$vlPosCR+Length(CR))
           Else
   // No delimiter found, get out of the loop
              $vtPasteboardData:=""
           End if
   // Repeat as long as we have data
        Until(Length($vtPasteboardData)=0)
     Else
        ALERT("The pasteboard does not any data that can be pasted as a record.")
  End case
End if
```

## System variables and sets

If the BLOB data is correctly appended to the pasteboard, OK is set to 1; otherwise OK is set to 0 and an error may be generated.

# CLEAR PASTEBOARD

CLEAR PASTEBOARD
Does not require any parameters

## Description

The **CLEAR PASTEBOARD** command clears the pasteboard of all its contents. If the pasteboard contains multiple instances of the same data, all instances are cleared. After a call to **CLEAR PASTEBOARD**, the pasteboard is empty.

You must call **CLEAR PASTEBOARD** once before appending new data to the pasteboard using the command **APPEND DATA TO PASTEBOARD**, because this latter command does not clear the pasteboard before appending the new data.

Calling **CLEAR PASTEBOARD** once and then calling **APPEND DATA TO PASTEBOARD** several times enables you to Cut or Copy the same data under different formats.

On the other hand, the **SET TEXT TO PASTEBOARD** and **SET PICTURE TO PASTEBOARD** commands automatically clear the pasteboard before appending the data to it.

## Example 1

The following code clears and then appends data to the pasteboard:

```
CLEAR PASTEBOARD ` Make sure the pasteboard is emptied
APPEND DATA TO PASTEBOARD("com.4d.private.picture.gif";$vxSomeData) ` Add some gif pictures
APPEND DATA TO PASTEBOARD("com.4d.private.text.rtf";$vxSylkData) ` Add some RTF text
```

## Example 2

See example for the **APPEND DATA TO PASTEBOARD** command.

# Get file from pasteboard

Get file from pasteboard ( xIndex ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| xIndex | Longint | ➡ | Xth file included in drag action |
| Function result | String | ⮌ | Pathname of file extracted from pasteboard |

## Description

The **Get file from pasteboard** command returns the absolute pathname of a file included in a drag and drop operation. Several files can be selected and moved simultaneously. The *xIndex* parameter is used to designate a file from among the set of files selected.

If there is no Xth file in the pasteboard, the command returns an empty string.

## Example

The following example can be used to retrieve in an array all the pathnames of the files included in a drag and drop operation:

```
ARRAY TEXT($filesArray;0)
C_TEXT($vfileArray)
C_INTEGER($n)
$n:=1
Repeat
   $vfileArray:=Get file from pasteboard($n)
   If($vfileArray#"")
      APPEND TO ARRAY($filesArray;$vfileArray)
      $n:=$n+1
   End if
Until($vfileArray="")
```

# GET PASTEBOARD DATA

GET PASTEBOARD DATA ( dataType ; data )

| Parameter | Type | | Description |
|---|---|---|---|
| dataType | String | ⇒ | Type of data to be extracted from pasteboard |
| data | BLOB | ⇐ | Requested data extracted from the pasteboard |

## Description

The **GET PASTEBOARD DATA** command returns, in the BLOB field or in the *data* variable, the data present in the pasteboard and whose type you pass in *dataType*. (If the pasteboard contains text copied within 4D, then the BLOB's character set is likely to be UTF-16.)

**Note:** In the context of copy/paste operations, the pasteboard corresponds to the clipboard.

In *dataType*, pass a value specifying the type of data to be retrieved. You can pass a 4D signature, a UTI type (Mac OS), a format name/number (Windows), or a 4-character type (compatibility). For more information about these types, please refer to the **Managing Pasteboards** section.

**Note:** You cannot read file type data with this command; in order to do this, you have to use the **Get file from pasteboard** command.

## Example

The following object methods for two buttons copy from and paste data to the array *asOptions* (pop-up menu, drop-downlist,...) located in a form:

```
 ` bCopyasOptions object method
If(Size of array(asOptions)>0) ` Is there something to copy?
   VARIABLE TO BLOB(asOptions;$vxClipData) ` Accumulate the array elements in a BLOB
   CLEAR PASTEBOARD ` Empty the pasteboard
   APPEND DATA TO PASTEBOARD("artx";asOptions) ` Note the data type arbitrarily chosen
End if

 ` bPasteasOptions object method
If(Pasteboard data size("artx")>0) ` Is there some "artx" data in the pasteboard?
   GET PASTEBOARD DATA("artx";$vxClipData) ` Extract the data from the pasteboard
   BLOB TO VARIABLE($vxClipData;asOptions) ` Populate the array with the BLOB data
   asOptions:=0 ` Reset the selected element for the array
End if
```

## System variables and sets

If the data is correctly extracted, OK is set to 1; otherwise OK is set to 0 and an error is generated.

# GET PASTEBOARD DATA TYPE

GET PASTEBOARD DATA TYPE ( 4Dsignatures ; nativeTypes {; formatNames} )

| Parameter | Type | | Description |
|---|---|---|---|
| 4Dsignatures | Text array | ⇦ | 4D signatures of data types |
| nativeTypes | Text array | ⇦ | Native data types |
| formatNames | Text array | ⇦ | Format names (Windows only), empty strings under Mac OS |

## Description

The **GET PASTEBOARD DATA TYPE** command gets the list of data types present in the pasteboard. This command should generally be used in the context of a drag and drop operation, within the On Drop or On Drag Over form events of the destination object. More particularly, it allows the pasteboard to be checked for the presence of a specific type of data.

This command returns the data types in several different forms via two (or three) arrays:

- The *4Dsignatures* array contains the data types expressed using the internal 4D signature (for example, “com.4d.private.picture.gif”). If a data type found is not recognized by 4D, an empty string (“”) is returned in the array.
- The *nativeTypes* array contains the data types expressed using their native types. The format of native types differs between Mac OS and Windows.
  - Under Mac OS, native types are expressed as UTIs (Uniform Type Identifier).
  - Under Windows, native types are expressed as numbers, with each number being associated with a format name. The *nativeTypes* array contains these numbers in the form of strings (“3”, “12”, and so on). If you want to use more explicit labels, it is recommended to use the optional *formatNames* array, which contains the format names of the native types under Windows.

  The *nativeTypes* array lets any type of data found in the pasteboard to be supported, including data whose type is not referenced by 4D.
- Under Windows, you can also pass the *formatNames* array, which receives the names of the data types found in the pasteboard. The values returned in this array can be used, for example, to build a format selection pop-up menu. Under Mac OS, the *formatNames* array returns empty strings.

For more information about the data types supported, please refer to the **Managing Pasteboards** section.

# GET PICTURE FROM PASTEBOARD

GET PICTURE FROM PASTEBOARD ( picture )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇦ | Picture extracted from pasteboard |

## Description

**GET PICTURE FROM PASTEBOARD** returns the picture present in the pasteboard in the *picture* field or variable.

**Note:** In the case of copy/paste operations, the pasteboard is equivalent to the Clipboard.

## Example

The following button's object method assigns the picture (jpeg or gif format) present in the pasteboard (if any) to the field [Employees]Photo:

```
If((Pasteboard data size("com.4d.private.picture.jpeg")>0)|(Pasteboard data size("com.4d.private.picture.gif")>0))
   GET PICTURE FROM PASTEBOARD([Employees]Photo)
Else
   ALERT("The pasteboard does not contain any pictures.")
End if
```

## System variables and sets

If the picture is correctly extracted, OK is set to 1; otherwise OK is set to 0.

# Get text from pasteboard

Get text from pasteboard -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String |  | Returns the text (if any) present in the pasteboard |

## Description

**Get text from pasteboard** returns the text present in the pasteboard.

**Note:** In the case of copy/paste operations, the pasteboard is equivalent to the Clipboard

If the pasteboard contains enriched text (for example in RTF format), the text will keep its attributes when it is dropped or pasted, if the destination area is compatible.

Note that 4D text fields and variables can contain up to 2 GB of text.

## System variables and sets

If the text is correctly extracted, OK is set to 1; otherwise OK is set to 0.

# Pasteboard data size

Pasteboard data size ( dataType ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| dataType | String | ➡ | Data type |
| Function result | Longint | ⮌ | Size (in bytes) of data located in the pasteboard or error code |

## Description

The **Pasteboard data size** command checks whether there is any data of the type you passed in *dataType* present in the pasteboard.

**Note:** In the case of copy/paste operations, the pasteboard is equivalent to the Clipboard.

If the pasteboard is empty or does not contain any data of the specified type, the command returns an error -102. If the pasteboard contains data of the specified type, the command returns the size of this data, expressed in bytes.

In *dataType*, pass a value specifying the type of data to be checked for. You can pass a 4D signature, a UTI type (Mac OS), a format name/number (Windows), or a 4-character type (compatibility). For more information about these types, please refer to the **Managing Pasteboards** section.

After you have detected that the pasteboard contains data of the type in which you are interested, you can extract that data from the pasteboard using one the following commands:

- If the pasteboard contains text type data, you can obtain that data using the **Get text from pasteboard** command, which returns a text value, or the **GET PASTEBOARD DATA** command, which returns the text in a BLOB.
- If the pasteboard contains picture type data, you can obtain that data using the **GET PICTURE FROM PASTEBOARD** command, which returns the picture in a picture field or variable, or the **GET PASTEBOARD DATA** command, which returns the picture in a BLOB.
- If the pasteboard contains a file pathname, you can extract it using the **Get file from pasteboard** command, which will return the file pathname.
- For any other data type, use the **GET PASTEBOARD DATA** command, which returns the data in a BLOB.

## Example 1

The following code tests whether the pasteboard contains a picture and, if so, copies that picture into a 4D variable:

```
If(Pasteboard data size(Picture data)=1) //Is there a picture in the pasteboard?
   GET PICTURE FROM PASTEBOARD($vPicVariable) //If so, extract the picture from the pasteboard
Else
   ALERT("There is no picture in the pasteboard.")
End if
```

## Example 2

Usually, applications cut and copy Text or Picture type data into the pasteboard, because most applications recognize these two standard data types. However, an application can append to the pasteboard several instances of the same data in different formats. For example, each time you cut or copy a part of a spreadsheet, the spreadsheet application could append the data under the hypothetical 'SPSH' format, as well as in SYLK and TEXT formats. The 'SPSH' instance would contain the data formatted using the application's data structure. The SYLK form would contain the same data, but using the SYLK format recognized by most of the other spreadsheet programs. Finally, the TEXT format would contain the same data, without the extra information included in the SYLK or the hypothetical 'SPSH' format. At this point, while writing Cut/Copy/Paste routines between 4D and that hypothetical spreadsheet application, assuming you know the description of the 'SPSH' format and that you are ready to parse SYLK data, you could write something like:

```
Case of
   ` First, check whether the pasteboard contains data from the hypothetical spreadsheet application
   :(Pasteboard data size('SPSH')>0)
   ` ...
   ` Second, check whether the pasteboard contains Sylk data
```

```
    :(Pasteboard data size('SYLK')>0)
  ` ...
  ` Finally check whether the pasteboard contains Text data
    :(Pasteboard data size('TEXT')>0)
  ` ...
End case
```

In other words, you try to extract from the pasteboard the instance of the data that carries most of the original information.

## Example 3

You want to drag some private data from different objects in your form. You can write:

```
 //source object
If(Form event=On Begin Drag Over)
   APPEND DATA TO PASTEBOARD("some.private.data";$data)
End if
```

```
 //target object
If(Form event=On Drag Over)
   $0:=Choose(Pasteboard data size("some.private.data")>0;0;-1)
End if
```

## Example 4

See the example for the **APPEND DATA TO PASTEBOARD** command.

# SET FILE TO PASTEBOARD

SET FILE TO PASTEBOARD ( file {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| file | String | ⇒ | File name or complete pathname of file |
| * | Operator | ⇒ | If passed = add; If omitted = replace |

## Description

The **SET FILE TO PASTEBOARD** command adds the complete pathname of the file passed in the *file* parameter. This command can be used to set up interfaces allowing the drag and drop of 4D objects to files on the desktop for example.

In the *file* parameter, you can pass either a complete pathname or a simple file name (without a pathname). In the latter case, the file must be located next to the database structure file.

The command accepts the star * as an optional parameter. By default, when this parameter is omitted, the command replaces the contents of the pasteboard by the last pathname specified by *file*. If you pass this parameter, the command adds the *file* to the pasteboard. This way it can contain a "stack" of file pathnames. In both cases, if data other than pathnames was present in the pasteboard, it is erased.

**Note:** The pasteboard is in read-only mode during the On Drag Over form event. It is therefore not possible to use this command in that context.

# SET PICTURE TO PASTEBOARD

SET PICTURE TO PASTEBOARD ( picture )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇒ | Picture to be placed in pasteboard |

## Description

**SET PICTURE TO PASTEBOARD** clears the pasteboard and puts a copy of the picture passed in *picture* into it.

**Note:** In the case of copy/paste operations, the pasteboard is equivalent to the Clipboard

The picture is transported in its native format (jpeg, tif, png, etc.).

After you have put a picture into the pasteboard, you can retrieve it using the **GET PICTURE FROM PASTEBOARD** command or for example **GET PASTEBOARD DATA**("com.4d.private.picture.gif";...).

## Example

Using a floating window, you display a form that contains the array *asEmployeeName*, which lists the names of the employees from an [Employees] table. Each time you click on a name, you want to copy the employee's picture to the pasteboard. In the object method for the array, you write:

```
If(asEmployeeName#0)
   QUERY([Employees];[Employees]Last name=asEmployeeName{asEmployeeName})
   If(Picture size([Employees]Photo)>0)
      SET PICTURE TO PASTEBOARD([Employees]Photo) ` Copy the employee's photo
   Else
      CLEAR PASTEBOARD ` No photo or no record found
   End if
End if
```

## System variables and sets

If a copy of the picture is correctly put into the pasteboard, the OK variable is set to 1.
If there is not enough memory to paste the picture into the pasteboard, the OK variable is set to 0, but no error is generated.

# SET TEXT TO PASTEBOARD

SET TEXT TO PASTEBOARD ( text )

| Parameter | Type | | Description |
|---|---|---|---|
| text | String | ⇨ | Text to be put into the pasteboard |

## Description

**SET TEXT TO PASTEBOARD** clears the pasteboard and then puts a copy of the text you passed in *text* into the pasteboard.

**Note:** In the case of copy/paste operations, the pasteboard is equivalent to the Clipboard

After you have put some text into the pasteboard, you can retrieve it using the **Get text from pasteboard** command or by calling for example **GET PASTEBOARD DATA** ("com.4d.private.text.native";...).

4D text expressions can contain up to 2 GB of text.

**Note:** The pasteboard is read only during the On Drag Over form event. It is not possible to use this command in this context.

## Example

See the example for the **APPEND DATA TO PASTEBOARD** command.

## System variables and sets

If a copy of the text is correctly placed in the pasteboard, the OK variable is set to 1. If there is not enough memory to place a copy of the text in the pasteboard, the OK variable is set to 0, but no error is generated.

# PHP

- Executing PHP scripts in 4D
- PHP Execute
- PHP GET FULL RESPONSE
- PHP GET OPTION
- PHP SET OPTION
- PHP modules support

# Executing PHP scripts in 4D

4D lets you directly execute PHP scripts. This new possibility gives you access to the wealth of utility libraries available via PHP. More particularly, these libraries provide the following functions:

- ciphering (MD5) and hashing,
- handling of ZIP files,
- handling of pictures,
- LDAP access,
- COM access (control of MS Office documents), etc.

This list is not exhaustive. For a complete list of the PHP modules available by default with 4D, please refer to the **PHP modules support** section. Also, please note that it is possible to install additional custom modules.

To execute a PHP script or function, you must use the **PHP Execute** command. By default, 4D includes version 5.4.11 of PHP. Executed scripts must be compatible with this version and with the modules installed.

For a complete description of PHP commands and syntax, please refer to the abundant PHP documentation available on the Internet. As an example, here are the addresses for a few reference sites:
http://us.php.net/manual/en/
http://phpdeveloper.org/
http://php.start4all.com/
http://php.resourceindex.com/Documentation/

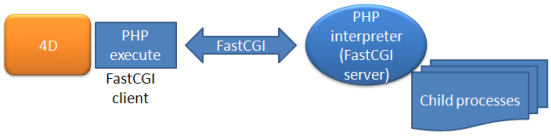
## Architecture

4D provides a PHP interpreter compiled in FastCGI, client-server type communication protocol between an application and a PHP interpreter.

The PHP interpreter pilots a set of system execution processes called "child processes". These processes are dedicated to the processing of requests sent by 4D. The execution of the requests is synchronous. For optimization reasons, by default up to five child processes can be run simultaneously (this number can be modified via the Database Settings or via the **SET DATABASE PARAMETER** command). Under Mac OS, these processes launched on the first request and are kept permanently by the PHP interpreter. Under Windows, 4D creates the processes according to its needs and recycles them if necessary. 4D automatically supports the handling of the processes of the PHP interpreter provided by default (launching and closing).

**Note:** If the 4D program quits unexpectedly while child PHP processes are still active, you must delete them manually via the system process management window.

The following diagram illustrates the 4D/PHP architecture of 4D:




This architecture works with a system of internal requests sent by 4D to a predefined TCP address (IP address and port number). If necessary, for example if several PHP interpreters are executed on the same machine, this address can be modified via the Database Settings or via the **SET DATABASE PARAMETER** command.

**Warning:** If you launch two 4D applications on the same machine and execute PHP statements on each of them (via the **PHP Execute** command), it is imperative to modify the listening ports of the FastCGI PHP interpreter so that you use a different one for each application. Otherwise, PHP statements may fail to execute correctly in an unpredictable manner and even freeze your 4D application.

## Using another PHP interpreter or another php.ini file

You can choose to use another PHP interpreter than the one provided by 4D. This allows you to keep the same PHP interpreter even if 4D is updated. In addition, it allows you to install all the custom modules you want -- in fact, you cannot

use a custom PHP file with the interpreter included in 4D. To use PHP configuration options other than the ones provided by default, you need to manage an external interpreter.

A custom PHP interpreter must respect two conditions:

- It must be compiled in FastCGI,
- It must be located on the same machine as 4D.

To use a custom PHP interpreter, you simply need to configure it so that it listens to a specific address and TCP port and to indicate to 4D to not activate the internal interpreter. These parameters can be specified either via database settings, or for the session via the **SET DATABASE PARAMETER** command. Of course, in this case, you must manage the starting and functioning of the interpreter yourself.

The *php.ini* initialization file is placed in the **Resources** folder of the database. The php.ini file can be used, more particularly, to declare the location of the PHP extensions.
If this file is not present during the first call, 4D will create it with the appropriate configuration options.

The php.ini file of the external interpreter must contain the following entries:

- **auto_prepend_file** which provides the complete pathname to the *4D_Execute_PHP.php* utility script. This script is found in [4D application]Resources/php/Windows or /Mac. Without this entry, only complete scripts can be executed: calls to a routine inside a script will not work.
- **display_errors** sets a "stderr" so that 4D can be informed when an error occurs during the execution of PHP code. Example:

```
; stderr - Display the errors to STDERR (only affects the CGI/CLI)
; To direct the errors to STDERR for the CGI/CLI:
display_errors = "stderr"
```

For more information about the configuration of custom php.ini files, please consult the comments found in the php.ini file provided by 4D.

# PHP Execute

PHP Execute ( scriptPath {; functionName {; phpResult {; param} {; param2 ; ... ; paramN}}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| scriptPath | Text | ⇒ | Pathname to PHP script or "" to execute a PHP function |
| functionName | Text | ⇒ | PHP function to be executed |
| phpResult | Operator, Variable, Field | ⇐ | Result of PHP function execution or * to not receive any result |
| param | Text, Boolean, Real, Longint, Date, Time | ⇒ | Parameter(s) of PHP function |
| Function result | Boolean | ⇒ | True = execution correct False = execution error |

## Description

The **PHP Execute** command can be used to execute a PHP script or function.

Pass the pathname of the PHP script to be executed in the *scriptPath* parameter. This can be a relative pathname if the file is located next to the database structure or an absolute path. The pathname can be expressed in either the system syntax or POSIX syntax.
If you want to execute a standard PHP function directly, pass an empty string ("") in *scriptPath*. The function name must be passed in the second parameter

Pass a PHP function name in the *functionName* parameter if you want to execute a specific function in the *scriptPath* script. If you pass an empty string or omit the *functionName* parameter, the script is entirely executed.

**Note:** PHP is case sensitive for function names. Do not use parentheses, just enter the function name only.

The *phpResult* parameter receives the result of the execution of the PHP function. You can pass either:

- a variable, an array or a field in order to receive the result,
- the * character if the function does not return any result or if you do not want to retrieve it.

The *phpResult* parameter can be of the Text, Longint, Real, Boolean, or Date type as well as (except for arrays) a field of the BLOB or Time type. 4D will carry out the conversion of the data and any adjustments needed according to the principles described in the **Conversion of data returned** section below.

- If you passed a function name in the *functionName* parameter, *phpResult* will receive what the PHP developer returned with the **return** command from the body of the function.
- If you use the command without passing a function name in the *functionName* parameter, *phpResult* will receive what the PHP developer returned with the **echo** command (or a similar command).

If you call a PHP function that expects arguments, use the *param1...N* parameters to pass one or more values. The values must be separated by semi-colons. You can pass values of the Alpha, Text, Boolean, Real, Integer, Longint, Date or Time type. Pictures, BLOBs and Objects are not accepted. You can send an array; in this case, you must pass a pointer to the array to the **PHP Execute** command, otherwise the current index of the array will be sent as an integer (see the example). The command accepts all types of arrays except for pointer, picture and 2D arrays.
The *param1...N* parameters are sent in PHP in the JSON format in UTF-8. They are automatically decoded with the PHP **json_decode** command before being sent to the PHP *functionName* function.

**Note:** For technical reasons, the size of parameters passed via the FastCGI protocol must not exceed 64 KB. You need to take this limitation into account if you use parameters of the Text type.

The command returns True if the execution has been carried out correctly on the 4D side, in other words, if the launching of the execution environment, the opening of the script and the establishing of the communication with the PHP interpreter were successful. Otherwise, an error is generated, which you can intercept with the **ON ERR CALL** command and analyze with **GET LAST ERROR STACK**.
In addition, the script itself may generate PHP errors. In this case, you must use the **PHP GET FULL RESPONSE** command in order to analyze the source of the error (see example 4).

**Note:** PHP can be used to configure error management. For more information, please refer, for example, to the following page: http://www.php.net/manual/en/errorfunc.configuration.php#ini.error-reporting.

### Conversion of data returned

The following table specifies how 4D interprets and converts data that is returned according to the type of the *phpResult* parameter.

| Type of *phpResult parameter* | Processing by 4D | Example |
|---|---|---|
| BLOB | 4D recovers the data received without any modifications(*). | |
| Text | 4D expects data encoded in UTF-8 (*). The PHP developer may need to use the PHP **utf8_encode** command. | Example of PHP script:<br>`echo utf8_encode(myText)` |
| Date | 4D expects a date sent as a string in the RFC 3339 format (sometimes called DATE_ATOM in PHP). This format is of the type "YYYY-MM-DDTHH:MM:SS", for example: 2005-08-15T15:52:01+00:00. 4D ignores the time part and returns the date in UTC. | |
| Time | 4D expects a time sent as a string in the RFC 3339 format (see the Date type). 4D ignores the date part and returns the number of seconds elapsed since midnight while considering the date in the local time zone. | Example of PHP script for sending 2h30'45":<br>`echo date( DATE_ATOM, mktime( 2,30,45))` |
| Integer or Real | 4D interprets the numerical value expressed with numbers, the + or - sign and/or the exponent prefixed by 'e'. Any '.' or ',' character is interpreted as a decimal separator. | Example of PHP script:<br>`echo -1.4e-16;` |
| Boolean | 4D returns True if it receives the string "true" from PHP or if the numerical evaluation gives a value that is not Null. | Example of PHP script:<br>`echo (a==b);` |
| Array | 4D considers that the PHP array was returned in the JSON format. | Example of PHP script for returning an array of two texts:<br>`echo json_encode( array( "hello", "world"));` |

(*) By default, HTTP headers are not returned:
- If you use **PHP Execute** by passing a function in the *functionName* parameter, HTTP headers are never returned in *phpResult*. They are only available through the **PHP GET FULL RESPONSE** command.
- If you use **PHP Execute** without a function name (the *functionName* parameter is omitted or contains an empty string), you can return HTTP headers by setting the PHP raw result option to True using the **PHP SET OPTION** command.

**Note:** If you need to recover large volumes of data using PHP, it is usually more efficient to pass by the *stdOut* buffer (**echo** command or similar) rather than by the function return. For more information, refer to the description of the **PHP GET FULL RESPONSE** command.

### Using environment variables

You can use the **SET ENVIRONMENT VARIABLE** command to specify the environment variables used by the script. Warning: after calling **LAUNCH EXTERNAL PROCESS** or **PHP Execute**, the set of environment variables is erased.

### Special functions

4D provides the following special functions:

- **quit_4d_php**: used to quit the PHP interpreter and all its child processes. If at least one child process is executing a script, the interpreter does not quit and the **PHP Execute** command returns False.
- **relaunch_4d_php**: used to relaunch the PHP interpreter.

Note that the interpreter is relaunched automatically when the first request is sent by **PHP Execute**.

## Example 1

Calling the "myPhpFile.php" script without any function. Here are the contents of the script:

```
<?php
echo 'Current PHP version: ' . phpversion();
?>
```

The following 4D code:

```
C_TEXT($result)
C_BOOLEAN($isOK)
$isOK:=PHP Execute("C:¥¥php¥¥myPhpFile.php";"";$result)
ALERT($Result)
```

... will display "Current PHP version: 5.3"

## Example 2

Calling the *myPhpFunction* function in the "myNewScript.php" script with parameters. Here are the contents of the script:

```
<?php
// . . . PHP code. . .
function myPhpFunction($p1, $p2) {
    return $p1 . ' '. $p2;
}
// . . . PHP code.. . .
?>
```

Calling with function:

```
C_TEXT($result)
C_TEXT($param1)
C_TEXT($param2)
C_BOOLEAN($isOk)
$param1 :="Hello"
$param2 :="4D world!"
$isOk:=PHP Execute("C:¥¥MyFolder¥¥myNewScript.php";"myPhpFunction";$result;$param1;$param2)
ALERT($result) // Displays "Hello 4D world!"
```

## Example 3

Quitting the PHP interpreter:

```
$ifOk:=PHP Execute("";"quit_4d_php")
```

## Example 4

Error management:

```
 // Installation of the error-management method
phpCommError:="" // Modified by PHPErrorHandler
$T_saveErrorHandler :=Method called on error
ON ERR CALL("PHPErrorHandler")</p><p> // Execution of script
C_TEXT($T_result)
If(PHP Execute("C:¥¥MyScripts¥¥MiscInfos.php";"";$T_result))
 // No error, $T_Result contains the result
Else
 // An error is detected, managed by PHPErrorHandler
   If(phpCommError="")
      ... // PHP error, use PHP GET FULL RESPONSE
   Else
      ALERT(phpCommError)
   End if
End if
```

```
 // Uninstalling method
ON ERR CALL($T_saveErrorHandler)
```

The ***PHP_errHandler*** method is as follows:

```
phpCommError:=""
GET LAST ERROR STACK(arrCodes;arrComps;arrLabels)
For($i;1;Size of array(arrCodes))
   phpCommError:=phpCommError+String(arrCodes{$i})+" "+arrComps{$i}+" "+
   arrLabels{$i}+Char(Carriage return)
End for
```

## Example 5

Dynamic creation by 4D of a script before its execution:

```
DOCUMENT TO BLOB("C:¥¥Scripts¥¥MyScript.php";$blobDoc)
If(OK=1)
   $strDoc:=BLOB to text($blobDoc;UTF8 text without length)

   $strPosition:=Position("function2Rename";$strDoc)

   $strDoc:=Insert string($strDoc;"_v2";Length("function2Rename")+$strPosition)

   TEXT TO BLOB($strDoc;$blobDoc;UTF8 text without length)
   BLOB TO DOCUMENT("C:¥¥Scripts¥¥MyScript.php";$blobDoc)
   If(OK#1)
      ALERT("Error on script creation")
   End if
End if
```

The script is then executed:

```
$err:=PHP Execute("C:¥¥Scripts¥¥MyScript.php";"function2Rename_v2";*)
```

## Example 6

Direct retrieval of a Date and Time type value. Here are the contents of the script:

```
<?php
// . . . code php. . .
echo date(DATE_ATOM, mktime(1, 2, 3, 4, 5, 2009));
// . . . code php. . .
?>
```

Receiving the date on the 4D side:

```
C_DATE($phpResult_date)
$result :=PHP Execute("C:¥php_scripts¥ReturnDate.php";"";$phpResult_date)
 //$phpResult_date is !05/04/2009 !

C_TIME($phpResult_time)
$result :=PHP Execute("C:¥php_scripts¥ReturnDate.php";"";$phpResult_time)
 //$phpResult_time is ?01 :02 :03 ?
```

## Example 7

Distribution of data in arrays:

```
ARRAY TEXT($arText ;0)
ARRAY LONGINT($arLong ;0)
$p1 :=","
```

```
$p2 :="11,22,33,44,55"
$phpok :=PHP Execute("";"explode";$arText;$p1;$p2)
$phpok :=PHP Execute("";"explode";$arLong;$p1;$p2)

 // $arText contains the Alpha values "11", "22", "33", etc.
 // $arLong contains the numbers, 11, 22, 33, etc.
```

## Example 8

Initialization of an array:

```
ARRAY TEXT($arText ;0)
$phpok :=PHP Execute("";"array_pad";$arText;->$arText;50;"undefined")
 // Execute in PHP: $arrTest = array_pad($arrTest, 50, ' undefined' );
 // Fills the $arText array with 50 "undefined" elements
```

## Example 9

Passing of parameters via an array:

```
ARRAY INTEGER($arInt;0)
$phpok :=PHP Execute("";"json_decode";$arInt;"[13,51,69,42,7]")
 // Execute in PHP: $arInt = json_decode(' [13,51,69,42,7]' );
 // Fills the array with the initial values
```

## Example 10

Here is an example of the basic use of the trim function, to remove extra spaces and/or invisible characters from the beginning and end of a string:

```
C_TEXT($T_String)
$T_String:="   Hello  "
C_BOOLEAN($B)
$B:=PHP Execute("";"trim";$T_String;$T_String)
```

For more information concerning the trim function, please refer to the PHP documentation.

# PHP GET FULL RESPONSE

PHP GET FULL RESPONSE ( stdOut {; errLabels ; errValues} {; httpHeaderFields {; httpHeaderValues}} )

| Parameter | Type | | Description |
|---|---|---|---|
| stdOut | Text variable, BLOB variable | ⇦ | Contents of stdOut buffer |
| errLabels | Text array | ⇦ | Labels of errors |
| errValues | Text array | ⇦ | Values of errors |
| httpHeaderFields | Text array | ⇦ | Names of HTTP headers |
| httpHeaderValues | Text array | ⇦ | Values of HTTP headers |

## Description

The **PHP GET FULL RESPONSE** command lets you obtain additional information about the response returned by the PHP interpreter. This command is particularly useful in the case of an error occurring during execution of the script.

The PHP script can write data in the stdOut buffer (echo, print, etc.). The command returns the data directly in the *stdOut* variable and applies the same conversion principles as described in the **PHP Execute** command.

The synchronized *errLabels* and *errValues* text arrays are filled when the execution of the PHP scripts causes errors. These arrays provide information in particular on the error origin, script and line. These two arrays are inseparable: if *errLabels* is passed, *errValues* must be passed as well.

Since exchanges between 4D and the PHP interpreter are carried out via FastCGI, the PHP interpreter functions as if it were called by an HTTP server and therefore sends HTTP headers. You can recover these headers and their values in the *httpHeaderFields* and *httpHeaderValues* arrays.

# PHP GET OPTION

PHP GET OPTION ( option ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ⇨ | Option to get |
| value | Text, Boolean | ⇦ | Current value of option |

## Description

The **PHP GET OPTION** command can be used to find out the current value of an option relating to the execution of PHP scripts.

Pass a constant from the "**PHP**" theme in the *option* parameter to designate the option to be gotten. The command returns the current value of the option in the *value* parameter. You can pass one of the following constants:

| Constant | Type | Value | Comment |
|---|---|---|---|
| PHP privileges | Longint | 1 | Definition of specific user privileges relating to the execution of the script.<br>**Possible value(s)**: String in the form "User:Password". For example: "root:jd51254d" |
| PHP raw result | Longint | 2 | Definition of processing mode for HTTP headers returned by PHP in the execution result when this result is of the Text type (when the result is of the BLOB type, headers are always kept).<br>**Possible value(s)**: Boolean. False (default value = remove HTTP headers from result. True = keep HTTP headers. |

**Note:** Only the user account is returned when you use the PHP privileges option with the **PHP GET OPTION** command (the password is not returned).

## Example

We want to find out the current user account:

```
C_TEXT($userAccount)
PHP GET OPTION(PHP privileges;$userAccount)
ALERT($userAccount)
```

# PHP SET OPTION

PHP SET OPTION ( option ; value {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ➡ | Option to be set |
| value | Text, Boolean | ➡ | New value of option |
| * | Operator | ➡ | If passed: modification only applied to next call |

## Description

The **PHP SET OPTION** command is used to set specific options before calling the **PHP Execute** command. The scope of this command is the current process.

Pass a constant from the "**PHP**" theme in the *option* parameter to designate the option to be modified and pass the new value for the option in the *value* parameter. Here is a description of the options:

| Constant | Type | Value | Comment |
|---|---|---|---|
| PHP privileges | Longint | 1 | Definition of specific user privileges relating to the execution of the script.<br>**Possible value(s)**: String in the form "User:Password". For example: "root:jd51254d" |
| PHP raw result | Longint | 2 | Definition of processing mode for HTTP headers returned by PHP in the execution result when this result is of the Text type (when the result is of the BLOB type, headers are always kept).<br>**Possible value(s)**: Boolean. False (default value = remove HTTP headers from result. True = keep HTTP headers. |

By default, **PHP SET OPTION** sets the option for all subsequent calls to **PHP Execute** of the process. If you want to set it for the next call only, pass the star (*) parameter.

## Example

Execute the "myAdminScript.php" script with Admin access rights:

```
PHP SET OPTION(PHP privileges;"admin:mypwd";*)
 `Since we pass the *, the admin privileges will only be used once
C_TEXT($result)
C_BOOLEAN($isOK)
$isOK:=PHP Execute("myAdminScript.php";$result)
If($isOK)
   ALERT($result)
End if
```

# PHP modules support

This appendix details the implementation of PHP modules in 4D. The following subjects are covered:

- List of standard PHP modules provided by default with the PHP interpreter of 4D
- List of standard PHP modules not retained by 4D
- Modifications of the php.ini file.

**Note:** If you want to install additional modules, you must use an external FastCGI-php interpreter (see **Using another PHP interpreter or another php.ini file**).

## Modules provided by default

The following table details the PHP modules provided by default with 4D.

### Generic modules

| Name | Web Site | Description |
| --- | --- | --- |
| BCMath | http://php.net/bc | Binary calculator handling numbers of any size and precision represented as strings.<br>Example:<br>`C_LONGINT($value;$result)`<br>`$value:=4`<br>`$ok:=PHP Execute("";"bcpow";$result;$value;3)` |
| Calendar | http://php.net/calendar | Set of functions simplifying conversion between different calendar formats. Based on Julian Day Count.<br>Example:<br>`C_LONGINT($NumberOfDays)`<br>`$ok:=PHP Execute("";"cal_days_in_month";$NumberOfDays;1;2;2010)` |
| Ctype | http://php.net/ctype | Functions that check whether a character or a string belongs to a certain character class, depending on the current local configuration<br>Example:<br>`// Check that all the characters of a string are punctuation marks`<br>`C_TEXT($myString)`<br>`$myString:=",.;/"`<br>`$ok:=PHP Execute("";"ctype_punct";$isPunct;$myString)` |
| Date and Time | http://php.net/datetime | Recovery of the date and time from the server where the PHP script is executed<br>Example:<br>`//Calculation of time of sunrise in Lisbon, Portugal`<br>`//Latitude: 38.4 North`<br>`//Longitude: 9 West`<br>`//Zenith ~= 90`<br>`//Time-lag: +1 GMT`<br>`C_TIME($SunriseTime)`<br>`$ok:=PHP Execute("";"date_sunrise";$SunriseTime;0;1;38,41;-9;90;1)` |
| DOM (Document Object Model) | http://php.net/dom | Use of XML documents via the DOM API in PHP 5 |
| Exif | http://php.net/exif | Work with image metadata. |
| Fileinfo(*) | http://php.net/fileinfo | Detection of type of contents and encoding of a file. |
| Filter | http://php.net/filter | Validate and filter data from a non-secure source, like user entries.<br>Example:<br>`C_LONGINT($filterId)`<br>`C_TEXT($result)`<br>`$ok:=PHP Execute("";"filter_id";$filterId;"validate_email")`<br>`$ok:=PHP Execute("";"filter_var";$result;"hop@123.com";$filterId)` |
| FTP (File Transfert Protocol) | http://php.net/ftp | Detailed access to a FTP server |
| Hash | http://php.net/hash | Message Digest engine. Allows direct or incremental processing of arbitrary length messages using a variety of hashing algorithms<br>Example:<br>`C_TEXT($md5Result)`<br>`$ok:=PHP Execute("";"md5";$md5Result;"this is my string to hash")` |
| GD | http://php.net/gd | Working with images |

| | | |
|---|---|---|
| (Graphics Draw) Library | | |
| Iconv | http://php.net/iconv | Conversion of files between various character sets |
| JSON (JavaScript Object Notation) | http://php.net/json | Implementation of JSON data exchange format |
| LDAP | http://php.net/ldap | LDAP is an access protocol to "folder servers" that store information in the form of a tree diagram |
| LibXML | http://php.net/libxml | Library of XML functions and constants |
| LibXSLT | http://php.net/xsl | Library of XSL transformation functions |
| Multibyte String | http://php.net/mbstring | Set of functions for working with strings that can be used to handle multi-byte character encodings or to convert character strings. |
| OpenSSL | http://php.net/openssl | Use of OpenSSL functions to generate and verify signatures, to seal (encode) and open (decode) data. |
| PCRE (Perl Compatible Regular Expressions) | http://php.net/pcre | Set of functions that implement rational expressions using the same syntax and semantics Perl 5<br><br>Example:<br>`// This example removes unnecessary spaces from a string`<br>`C_TEXT($myString)`<br>`$myString:="foo o bar"`<br>`PHP Execute("";"preg_replace";$myString;"/¥¥s¥¥s+/";" ";$myString)`<br>`ALERT($myString)`<br>`//The string is now 'foo o bar' without repeated spaces` |
| PDO (PHP Data Objects) | http://php.net/pdo | Interface for accessing a database. Requires a database-specific PDO driver. |
| PDO_SQLITE | http://php.net/pdo_sqlite | Driver that implements the PHP Data Objects (PDO) interface to allow PHP access to SQLite 3 databases. |
| Reflection | http://php.net/reflection | Complete reflection API that lets you reverse-engineer classes, interfaces, functions and methods as well as extensions |
| Phar (PHP Archive) | http://php.net/phar | Allows a complete PHP application to be included in a unique file named "phar" (PHP Archive) to facilitate its installation and configuration |
| Session | http://php.net/session | Support of PHP sessions<br>Example:<br>Sessions are used in Web applications to preserve the context between each request. When you call **PHP Execute** in 4D, the PHP script can start a session and store anything that is useful to be kept as context in the associated $_SESSION array. If a PHP script uses sessions, you must obtain the session ID returned by PHP using the **PHP GET FULL RESPONSE** command and specify it before each call to **PHP Execute** using the **SET ENVIRONMENT VARIABLE** command.<br>`// "PHP Execute with context" method`<br>`If(<>PHP_Session#"")`<br>`SET ENVIRONMENT VARIABLE("HTTP_COOKIE";<>PHP_Session)`<br>`End if`<br>`If(PHP Execute($1))`<br>`PHP GET FULL RESPONSE($0;$errorInfos;$errorValues;$headerFields;$headerValues)`<br>`$idx:=Find in array($headerFields;"Set-Cookie")`<br>`If($idx>0)`<br>`<>PHP_Session:=$headerValues{$idx}`<br>`End if`<br>`End if` |

| | | |
|---|---|---|
| SimpleXML | http://php.net/simpleXML | Very simple and easy-to-use tools to be used to convert XML to an object that can be processed with its properties and array iterators |
| Sockets | http://php.net/sockets | Implementation of a low-level interface to the socket communication functions based on BSD sockets and providing the possibility to act as both a socket server as well as a client. |
| SPL (Standard PHP Library) | http://php.net/spl | Collection of interfaces and classes that are meant to solve standard problems. |
| SQLite | http://php.net/sqlite | Extension for the SQLite database engine. This engine is embeddable. |
| SQLite3 | http://php.net/sqlite3 | Support for SQLite version 3 databases |
| Tokenizer | http://php.net/tokenizer | Functions that let you write your own PHP analysis tools or modification tools without having to deal with the language specification at the lexical level |
| XML (eXtensible Markup Language) | http://php.net/xml | Parsing of XML documents |
| XMLreader | http://php.net/xmlreader | XML Pull parser |
| XMLwriter | http://php.net/xmlwriter | Generation of XML format streams or files |
| Zlib | http://php.net/zlib | Reading and writing of gzip (.gz) compressed files<br>Example:<br>`WEB GET HTTP HEADER($names;$values)`<br>`$pos:=Find in array($names;"Accept-Encoding")`<br>`If($pos>0)`<br>`  Case of`<br>`  :(Position($values{$pos};"gzip")>0)`<br>`  WEB SET HTTP HEADER("Content-Encoding: gzip")`<br>`  PHP Execute("";"gzencode";$html;$html)`<br>`  :(Position($values{$pos};"deflate")>0)`<br>`  WEB SET HTTP HEADER("Content-Encoding: deflate")`<br>`  PHP Execute("";"gzdeflate";$html;$html)`<br>`  End case`<br>`End if`<br>`WEB SEND TEXT($html)` |
| Zip | http://php.net/zip | Reading and writing of ZIP compressed archives and the files inside them |

(*) In the current version of 4D, these modules are not available under Windows

### Modules only available under Windows

For structural reasons, the following PHP modules are only available on the Windows platform.

| Name | Web Site | Description |
|---|---|---|
| COM & .NET | http://php.net/com | COM (Component Object Model) is one of the main ways for applications and components to communicate on Windows platforms. In addition, 4D supports the instantiation and creation of .Net assemblies via the COM layer. |
| ODBC (Open DataBase Connectivity) | http://php.net/odbc | In addition to standard ODBC support, the Unified ODBC functions in PHP gives you access to several databases that have borrowed the semantics of the ODBC API to implement their own API. |
| WDDX (Web Distributed Data eXchange) | http://php.net/wddx | Facilitates data exchanges between Web applications over the Web, regardless of the platform |

## Disabled modules

The following PHP modules have not been implemented in 4D. The rightmost column gives the reason they were not implemented:

| Name | Web Site | Cause - Alternative solution |
| --- | --- | --- |
| Mimetype | http://php.net/mime-magic | Obsolete (Deprecated) - Use Fileinfo |
| POSIX (Portable Operating System Interface) | http://php.net/posix | Obsolete (Deprecated) |
| Regular Expression (POSIX Extended) | http://php.net/regex | Obsolete (Deprecated) - Use PCRE |
| Crack | http://php.net/crack | Restrictive license |
| ffmpeg | http://ffmpeg-php.sourceforge.net/ | Restrictive license - Use ffmpeg in command line with LAUNCH EXTERNAL PROCESS |
| Image Magick | http://php.net/manual/book.imagick.php | Restrictive license - Use GD 2 |
| IMAP (Internet Message Access Protocol) | http://php.net/imap | Restrictive license - Use the 4D Internet Commands integrated plugin |
| PDF (Portable Document Format) | http://php.net/pdf | Restrictive license - Use Haru PDF |
| Mysqlnd (MySQL Native Driver) | http://dev.mysql.com/downloads/connector/php-mysqlnd/ | Not pertinent in the 4D environment |

## Customizing the php.ini file

The "php.ini" file to modify (see below) can be located either in the **Resources¥php** folder of the 4D application (default file) or in the **Resources** folder of the database (custom file). For more on this subject, refer to **Executing PHP scripts in 4D**.

**Warning:** Modifying the "php.ini" file must be done with caution and requires a good knowledge of PHP. For more information about the configuration of custom php.ini files, you can consult the comments found in the php.ini file provided by 4D.

**Note:** If the duration of PHP processing is relatively long (beyond 30 seconds), by default a 'timeout' error will be returned in 4D and the processing will fail. In this case, you can set the default *timeout* in order to allocate more time to PHP execution. There are two ways to do this:

- by setting the **max_execution_time** variable in the "php.ini" file (pass a value in seconds). Warning: this setting affects all the scripts.
- by calling the **set_time_limit(nbSec)** command in the PHP execution script that is performing the long processing. Pass the maximum duration allocated to the execution of the PHP script in *nbSec*. We recommend using this setting since it only affects this script. Usually, for security reasons, it is preferable to keep a lower *timeout* value for PHP scripts.

# Pictures

- Pictures
- BLOB TO PICTURE
- COMBINE PICTURES
- CONVERT PICTURE
- CREATE THUMBNAIL
- Equal pictures
- Get picture file name
- GET PICTURE FORMATS New 16.0
- GET PICTURE FROM LIBRARY
- GET PICTURE KEYWORDS
- GET PICTURE METADATA
- Is picture file
- PICTURE CODEC LIST
- PICTURE LIBRARY LIST
- PICTURE PROPERTIES
- Picture size
- PICTURE TO BLOB
- PICTURE TO GIF
- READ PICTURE FILE
- REMOVE PICTURE FROM LIBRARY
- SET PICTURE FILE NAME
- SET PICTURE METADATA
- SET PICTURE TO LIBRARY
- TRANSFORM PICTURE
- WRITE PICTURE FILE
- *_o_PICTURE TYPE LIST*
- *_o_QT COMPRESS PICTURE*
- *_o_QT COMPRESS PICTURE FILE*
- *_o_QT LOAD COMPRESS PICTURE FROM FILE*
- *_o_SAVE PICTURE TO FILE*

# Pictures

## Native Formats Supported

4D integrates native management of picture formats. This means that pictures will be displayed and stored in their original format, without any interpretation in 4D. The specific features of the different formats (shading, transparent areas, etc.) will be retained when they are copied and pasted, and will be displayed without alteration. This native support is valid for all pictures stored in 4D: library pictures, pictures pasted into forms in Design mode, pictures pasted into fields or variables in Application mode, etc.

4D uses native APIs to encode and decode pictures (fields and variables) under both Windows and Mac OS. These implementations provide access to numerous native forms, including the RAW format, currently used by digital cameras.

- **Under Windows**, 4D uses WIC (Windows Imaging Component). WIC natively supports the following formats: BMP, PNG, ICO (decoding only), JPEG, GIF, TIFF and WDP (Microsoft Windows Digital Photo).
  It is possible to use additional formats such as JPEG-2000 by installing third-party WIC codecs.
- **Under Mac OS**, 4D uses ImageIO. All the available ImageIO codecs are therefore natively supported for decoding (reading) as well as encoding (writing):

| Decoding | Encoding |
|---|---|
| public.jpeg | public.jpeg |
| com.compuserve.gif | com.compuserve.gif |
| public.png | public.png |
| public.jpeg-2000 | public.jpeg-2000 |
| com.nikon.raw-image | public.tiff |
| com.pentax.raw-image | com.adobe.photoshop.image |
| com.sony.arw-raw-image | com.adobe.pdf |
| com.adobe.raw-image | com.microsoft.bmp |
| public.tiff com.canon.crw-raw-image | com.truevision.tga-image |
| com.canon.cr2-raw-image | com.sgi.sgi-image |
| com.canon.tif-raw-image | com.apple.pict *(deprecated)* |
| com.sony.raw.image | com.ilm.openexr-image |
| com.olympus.raw-image | |
| com.konicaminolta.raw-image | |
| com.panasonic.raw-image | |
| com.fuji.raw-image | |
| com.adobe.photoshop-image | |
| com.adobe.illustrator.ai-image | |
| com.adobe.pdf | |
| com.microsoft.ico | |
| com.microsoft.bmp | |
| com.truevision.tga-image | |
| com.sgi.sgi-image | |
| com.apple.quicktime-image *(deprecated)* | |
| com.apple.icns | |
| com.apple.pict *(deprecated)* | |
| com.apple.macpaint-image | |
| com.kodak.flashpix-image | |
| public.xbitmap-image | |
| com.ilm.openexr-image | |
| public.radiance | |

Under Windows as under Mac OS, the formats supported vary according to the operating system and the custom codecs that are installed on the machines. To find out which codecs are available, you must use the **PICTURE CODEC LIST** command.

**Note:** WIC and ImageIO permit the use of metadata in pictures. Two commands, **SET PICTURE METADATA** and **GET PICTURE METADATA**, let you benefit from metadata in your developments.

### Picture Codec IDs

Picture formats recognized by 4D are returned by the **PICTURE CODEC LIST** command as picture Codec IDs. They can be returned in the following forms:

- As an extension (for example ".gif")
- As a Mime type (for example "image/jpeg")

The form returned for each format will depend on the way the Codec is recorded at the operating system level.
Most of the 4D picture management commands can receive a Codec ID as a parameter. It is therefore imperative to use the system ID returned by the **PICTURE CODEC LIST** command.

### Unavailable picture format

A specific icon is displayed for pictures saved in a format that is not available on the machine. The extension of the missing format is shown at the bottom of the icon:


The icon is automatically used wherever the picture is meant to be displayed:


This icon indicates that the picture cannot be displayed or manipulated locally -- but it can be saved without alteration so that it can be displayed on other machines. This is the case, for instance, for PDF pictures on Windows, or for pictures based on PICT displayed on a 64-bit 4D Server under OS X.

### Activation of QuickTime (compatibility)

By default, the picture codecs related to QuickTime are no longer supported in 4D beginning with v14.

For compatibility reasons, you can reactivate QuickTime in your application by means of the QuickTime support option of the **SET DATABASE PARAMETER** command. However, we no longer recommend using QuickTime codecs.

**Note:** The QuickTime reactivation option is ignored in 64-bit versions of 4D Developer Edition (no QuickTime support).

## Mouse Coordinates in a Picture

4D lets you retrieve the local coordinates of the mouse in a picture field or variable in case of a click or a hovering, even if a scroll or zoom has been applied to the picture. This mechanism, similar to that of a picture map, can be used, for example, to handle scrollable button bars or the interface of cartography software.

The coordinates are returned in the *MouseX* and *MouseY* **System Variables**. The coordinates are expressed in pixels with respect to the top left corner of the picture (0,0). If the mouse is outside of the picture coordinates system, -1 is returned in *MouseX* and *MouseY*.

You can get the value of these variables as part of the On Clicked, On Double Clicked, On Mouse up, On Mouse Enter, or On Mouse Move form events.

## Picture Operators

4D allows you to carry out **operations** on 4D pictures, such as concatenation, superimposing, etc. This point is covered in the **Picture Operators** section.

# BLOB TO PICTURE

BLOB TO PICTURE ( pictureBlob ; picture {; codec} )

| Parameter | Type | | Description |
|---|---|---|---|
| pictureBlob | BLOB | ⇒ | BLOB containing a picture |
| picture | Picture | ⇐ | Picture from BLOB |
| codec | String | ⇒ | Picture codec ID |

## Description

The **BLOB TO PICTURE** command inserts a picture stored in a BLOB into a 4D picture variable or field, regardless its original format.

This command is similar to the command **READ PICTURE FILE**, it just applies to a BLOB instead of a file. It allows you to display pictures stored in native format into BLOBs. You can load a picture into a BLOB using, for example, the command **DOCUMENT TO BLOB** or **PICTURE TO BLOB**.

A BLOB variable or field containing a picture is passed in the *pictureBlob* parameter. The picture can be in any format supported natively by 4D. You can obtain the list of available formats using the **PICTURE CODEC LIST** command. If you pass the optional *codec* parameter, 4D will use the value provided in this parameter to decode the BLOB (see the specific functioning of the command with this third parameter below).

Pass in the *picture* parameter the 4D picture field or variable which should display the picture.

**Note:** The internal picture format will be stored within the 4D variable or field.

Once the command has been executed, if the BLOB was correctly decoded, the *picture* parameter contains the picture to display.

The optional *codec* parameter lets you specify the codec to be used for decoding the BLOB.

If you pass a codec recognized by 4D in *codec* (returned by the **PICTURE CODEC LIST** command), it is applied to the BLOB and the picture is returned in the *picture* field or variable.

If you pass a codec that is not recognized by 4D in *codec*, a new codec is recorded dynamically with the ID passed in the parameter. 4D then returns a picture that encapsulates the BLOB and the OK variable is set to 1. In this case, to retrieve the BLOB, you will need to use the **PICTURE TO BLOB** command with the same custom ID. This particular mechanism can be used to meet two specific needs:

- encapsulation of a BLOB (that is not a picture) into a picture,
- loading a picture without using a codec.

The implementation of these mechanisms allows, more specifically, the creation of "BLOB arrays" via picture arrays. This technique must be used with caution because, since the arrays are loaded entirely into memory, working with large sized BLOBs can affect the functioning of the application.

**Note:** A BLOB created by the **VARIABLE TO BLOB** command is managed automatically; it is not necessary to pass a codec to encapsulate it since the BLOB is "signed." In this case, for the opposite operation, you will need to pass ".4DVarBlob" to the **PICTURE TO BLOB** command as the codec ID.

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. If the conversion has failed (QuickTime is not installed, the BLOB does not contain a readable picture, the codec parameter recognized but BLOB not validated, etc.), OK is set to 0 and the 4D picture variable or field is returned empty.

# COMBINE PICTURES

COMBINE PICTURES ( resultingPict ; pict1 ; operator ; pict2 {; horOffset ; vertOffset} )

| Parameter | Type | | Description |
|---|---|---|---|
| resultingPict | Picture | ⇐ | Picture resulting from combination |
| pict1 | Picture | ⇒ | First picture to combine |
| operator | Longint | ⇒ | Type of combination to be done |
| pict2 | Picture | ⇒ | Second picture to combine |
| horOffset | Longint | ⇒ | Horizontal offset for superimposition |
| vertOffset | Longint | ⇒ | Vertical offset for superimposition |

## Description

The **COMBINE PICTURES** command combines the *pict1* and *pict2* pictures in *operator* mode in order to produce a third, *resultingPict*. The resulting picture is of the compound type and keeps all the characteristics of the source pictures.

**Note:** This command extends the functionalities offered by the conventional picture combination operators (+/, etc., see the **Picture Operators** section). These operators remain entirely usable in 4D v11.

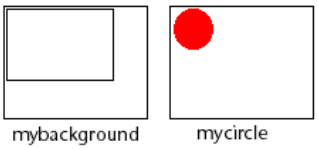
In *operator*, pass the type of combination to be applied. Three types of combinations, which can be accessed via the constants of the "**Picture Transformation**" theme, are proposed:

- Horizontal concatenation (1): *pict2* is attached to *pict1*, the top left corner of *pict2* coincides with the top right corner of *pict1*.
- Vertical concatenation (2): *pict2* attached to *pict1*, the top left corner of *pict2* coincides with the lower left corner of *pict1*.
- Superimposition (3): *pict2* is placed over *pict1*, the top left corner of *pict2* coincides with the top left corner of *pict1*. If the optional *horOffset* and *vertOffset* parameters are used, a translation is applied to *pict2* before superimposition. The values passed in *horOffset* and *vertOffset* must correspond to pixels. Pass positive values for an offset to the right or towards the bottom and a negative value for an offset to the left or towards the top.

**Note:** Superimposition carried out by the **COMBINE PICTURES** command differs from the superimposition provided by the conventional & and |operators (exclusive and inclusive superimposition). While the **COMBINE PICTURES** command preserves the characteristics of each source picture in the resulting picture, the & and | operators process each pixel and generate a bitmap picture in all cases. These operators, originally intended for black and white pictures, are now obsolete.

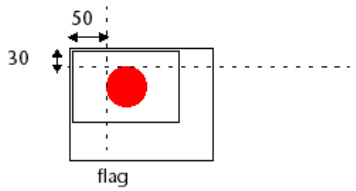
## Example

Given the following pictures:




```
COMBINE PICTURES(flag;mybackground;Superimposition;mycircle;50;30)
```

Result:



# CONVERT PICTURE

CONVERT PICTURE ( picture ; codec {; compression} )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇨ | Picture to be converted |
| | | ⇦ | Converted picture |
| codec | String | ⇨ | Picture Codec ID |
| compression | Real | ⇨ | Quality of compression |

## Description

The **CONVERT PICTURE** command converts *picture* into a new type.

The *codec* parameter indicates the type of picture to be generated. A Codec can be an extension (for example, ".gif") or a Mime type (for example, "image/jpeg"). You can get a list of Codecs that are available using the **PICTURE CODEC LIST** command.

If the *picture* field or variable is a compound type (if, for example, it is the result of a copy-paste action), only the information corresponding to the codec type are preserved in the resulting picture.

**Note:** If the type of *codec* requested is the same as the original type of the *picture*, no conversion is carried out and the picture is returned "as is" (except when the *compression* parameter is used, see below).

The optional *compression* parameter, if passed, can be used to specify the compression quality to be applied to the resulting picture when a compatible Codec is used. In *compression*, pass a value between 0 and 1 to specify the quality of the compression, where 0 is the most mediocre quality (high compression) and 1 the best quality (low compression). This parameter is only taken into account when the Codec supports compression (for example JPEG or HDPhoto) and is supported by the WIC and ImageIO APIs. For more information about picture management APIs in 4D, please refer to the **Pictures** section. By default, if you omit the *compression* parameter, the best quality is applied (compression =1).

## Example 1

Conversion of the vpPhoto picture to the jpeg format:

```
CONVERT PICTURE(vpPhoto;".jpg")
```

## Example 2

Conversion of a picture with 60% quality:

```
CONVERT PICTURE(vPicture;".JPG";0.6)
```

# CREATE THUMBNAIL

CREATE THUMBNAIL ( source ; dest {; width {; height {; mode {; depth}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| source | Picture | ⇒ | 4D picture field or variable to convert as a thumbnail |
| dest | Picture | ⇐ | Resulting thumbnail |
| width | Integer | ⇒ | Thumbnail width in pixels, Default value = 48 |
| height | Integer | ⇒ | Thumbnail height in pixels, Default value = 48 |
| mode | Integer | ⇒ | Thumbnail creation mode Default value = Scaled to fit prop centered (6) |
| depth | Integer | ⇒ | Obsolete, do not use |

## Description

The **CREATE THUMBNAIL** command returns a thumbnail from a given source picture. Thumbnails are usually used for picture preview within multimedia software or Web sites.

You pass in the *source* parameter the 4D variable or field containing the picture to reduce to a thumbnail. You pass in the *dest* parameter the 4D picture field or variable which should host the resulting thumbnail.

The optional parameters *width* and *height* define the required thumbnail size (in pixels). If you omit these parameters, the thumbnail default size will be 48 x 48 pixels.

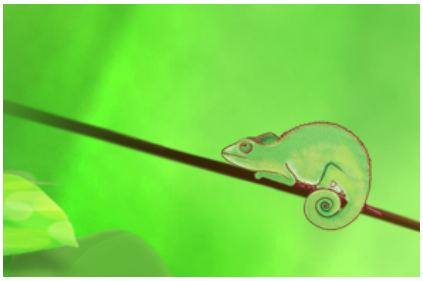
The optional parameter *mode* defines the thumbnail creation mode, i.e. the resizing mode. Three modes are available. The following predefined constants are provided by 4D in the "**Picture Display Formats**" constant theme:

| Constant | Type | Value |
|---|---|---|
| Scaled to fit | Longint | 2 |
| Scaled to fit prop centered | Longint | 6 |
| Scaled to fit proportional | Longint | 5 |

**Note:** Only these constants can be used with **CREATE THUMBNAIL**. The other constants in this theme cannot be applied to this command.

If you do not enter any parameter, the "Scaled to fit prop centered" mode (6) is applied by default. Below is an illustration of the various modes:

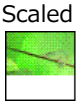
Source picture



Resulting thumbnails (48x48)

- Scaled to fit = 2

- Scaled to fit proportional = 5



- Scaled to fit prop centered = 6 (default mode)

**Note:** With the "Scaled to fit proportional" and the "Scaled to fit prop centered", the free space will be displayed in white. When these modes are applied to picture field or variable in 4D forms, the free space is transparent.

The *depth* parameter is ignored and must be omitted. The command always uses the current screen depth (number of colors).

# Equal pictures

Equal pictures ( picture1 ; picture2 ; mask ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| picture1 | Picture field, Picture variable | ⇒ | Original source picture |
| picture2 | Picture field, Picture variable | ⇒ | Picture to compare |
| mask | Picture field, Picture variable | ⇐ | Resulting mask |
| Function result | Boolean | ⇒ | True if both pictures are identical; otherwise, False |

## Description

The **Equal pictures** command precisely compares both the dimensions and the contents of two pictures.

Pass the source picture in *picture1* and the picture you want to compare with it in *picture2*.

- If the pictures are not the same dimension, the command returns **False** and the *mask* parameter contains a blank picture.
- If the pictures are of the same dimension but with different contents, the command returns **False** and the *mask* parameter contains the resulting picture mask based on a comparison of the two pictures. This comparison is performed pixel by pixel, and each pixel that does not match appears white on a black background.
- If both pictures are exactly the same, the command returns **True** and the *mask* parameter contains a picture that is completely black.

## System variables and sets

If the command is executed successfully (the two pictures are compared), the system variable OK is set to 1. In the case of an anomaly, particularly if one of the pictures is not initialized (blank picture), the OK variable is set to 0.

## Example

In the following example, we compare two pictures (pict1 and pict2) and display the resulting mask:


Here is the code for the **Compare** button:

```
$equal :=Equal pictures($pict1;$pict2;$mask)
```

# Get picture file name

Get picture file name ( picture ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture field, Picture variable | ⇒ | Picture for which to get default name |
| Function result | Text | ⇐ | Default name of picture file |

## Description

The **Get picture file name** command returns the current default name of the picture passed as parameter.

The default name is used when exporting the picture to a disk file. It can be set automatically based on the original name of the picture file imported into the picture field or variable, or using the **SET PICTURE FILE NAME** command. For more information, refer to the *Design Reference* manual.

If the picture does not have a default name, the command returns an empty string.

# GET PICTURE FORMATS

GET PICTURE FORMATS ( picture ; codecIDs )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇨ | Picture field or variable to analyze |
| codecIDs | Text array | ⇦ | Picture codec IDs |

## Description

The **GET PICTURE FORMATS** command returns an array of all the codec IDs (picture formats) contained in the *picture* passed as parameter. A 4D picture (field or variable) can contain the same picture encoded in different formats, such as PNG, BMP, GIF, etc.

In the *picture* parameter, you pass a picture field or a picture variable whose included formats you want to be returned in the *codecIDs* array.

The codec IDs returned are established by 4D in exactly the same way as for the **PICTURE CODEC LIST** command. They can be returned in the following forms:

- As extensions (for example, ".gif")
- As Mime types (for example, "image/jpeg")
- As 4-character QuickTime codes

**Notes:**

- The following codecs, handled internally by 4D, are always returned as extensions: JPEG, PNG, TIFF, GIF, BMP, SVG, PDF, EMF.
- 4-character QuickTime codes may be returned in databases where the QuickTime support compatibility option has been set (using the **SET DATABASE PARAMETER** command). However, QuickTime is no longer supported in 4D and we do not recommend using QuickTime codecs.

For more information about picture codec IDs, refer to the **Pictures** section.

## Example

You want to know the picture formats stored in a field for the current record:

```
ARRAY TEXT($aTPictureFormats;0)
 //Get all the formats saved
GET PICTURE FORMATS([Employees]Photo;$aTPictureFormats)
```

# GET PICTURE FROM LIBRARY

GET PICTURE FROM LIBRARY ( picRef | picName ; picture )

| Parameter | Type | | Description |
|---|---|---|---|
| picRef \| picName | Longint, String | ⇨ | Reference number of Picture Library graphic or Name of Picture Library graphic |
| picture | Picture variable | ⇦ | Picture from the Picture Library |

## Description

The **GET PICTURE FROM LIBRARY** command returns in the *picture* parameter the Picture Library graphic whose reference number is passed in *picRef* or whose name is passed in *picName*.

If there is no picture with that reference number or name, **GET PICTURE FROM LIBRARY** leaves *picture* unchanged.

## Example 1

The following example returns in *vgMyPicture* the picture whose reference number is stored in the local variable *$vlPicRef*:

```
GET PICTURE FROM LIBRARY($vlPicRef;vgMyPicture)
```

## Example 2

The following example returns in *$DDcom_Prot_MyPicture* the picture with the name "DDcom_Prot_Button1" stored in the Picture Library:

```
GET PICTURE FROM LIBRARY("DDcom_Prot_Button1";$DDcom_Prot_MyPicture)
```

## Example 3

See the third example for the **PICTURE LIBRARY LIST** command.

## System variables and sets

If the Picture Library exists, the **OK** variable is set to 1. Otherwise, **OK** is set to zero.

## Error management

If there is not enough memory to return the picture, an error -108 is generated. You can catch this error using an error-handling method.

# GET PICTURE KEYWORDS

GET PICTURE KEYWORDS ( picture ; arrKeywords {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture field, Picture variable | ⇒ | Picture for which to get associated keywords |
| arrKeywords | Text array | ⇐ | Array containing extracted keywords |
| * | Operator | ⇒ | If passed = use distinct values |

## Description

The **GET PICTURE KEYWORDS** command returns, in the *arrKeywords* array, the list of keywords associated with the picture passed as parameter.

Only keywords set using **IPTC/Keywords** metadata are taken into account. Other types of metadata are ignored by the command. The command only works with picture types that support this type of metadata (JPEG, TIFF, etc.).

**Note:** Metadata of the IPTC/Keywords type are now indexable in 4D (see the *Design Reference* manual).

If you pass the * parameter, the method only returns "distinct values" of keywords, in other words, a list with no duplicates.

If the picture does not contain keywords or IPTC/Keywords metadata, the command returns an empty array and no error is generated.

**Note:** Results returned by this command can differ according to the current value of the "Consider only non-alphanumeric chars for keywords" database setting (see the **Database/Data storage page**).

# GET PICTURE METADATA

GET PICTURE METADATA ( picture ; metaName ; metaContents {; metaName2 ; metaContents2 ; ... ; metaNameN ; metaContentsN} )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ➡ | Picture whose metadata you want to get |
| metaName | Text | ➡ | Name or path of block to get |
| metaContents | Variable | ⬅ | Metadata contents |

## Description

The **GET PICTURE METADATA** command can be used to read the contents of the metadata (or meta-tags) found in *picture* (4D picture field or variable). For more information about metadata, please refer to the description of the **SET PICTURE METADATA** command.

In the *metaName* parameter, pass a string specifying the type of metadata to retrieve. You can pass:

- a constant from the **Picture Metadata Names** theme containing a tag path,
- the name of a complete block of metadata ("TIFF", "EXIF", "GPS" or "IPTC"),
- an empty string ("").

Pass the variable intended to receive the metadata in the *metaContents* parameter.

- If you passed a tag path in *metaName*, the *metaContents* parameter will directly contain the value to get. The value will be converted to the type of the variable. Variables of the Text type will be formatted in XML (XMP standard). You can pass an array when the metadata contains more than one value (this is the case, more particularly, for the IPTC Keywords tags).
- If you passed a block name or an empty string in *metaName*, the *metaContents* parameter must be a valid XML DOM element reference. In this case, the contents of the designated block (or all the blocks if you passed an empty string in *metaName*) is recopied into the element referenced.

## Example 1

Use of DOM tree structures

```
$xml:=DOM Create XML Ref("Root") //Creation of an XML DOM tree

 //Reception of TIFF metadata
$_Xml_TIFF:=DOM Create XML element($xml;"/Root/TIFF")
GET PICTURE METADATA(vPicture;"TIFF";$_Xml_TIFF)

 //Reception of GPS metadata
$_Xml_GPS:=DOM Create XML element($xml;"/Root/GPS")
GET PICTURE METADATA(vPicture;"GPS";$_Xml_GPS)
```

## Example 2

Use of variables

```
C_DATE($dateAsDate)
GET PICTURE METADATA(vPicture;TIFF date time;$dateAsDate)
 //only returns the date since $dateAsDate is of the Date type

C_TEXT($dateAsText)
GET PICTURE METADATA(vPicture;TIFF date time;$dateAsText)
 //only returns the date but in XML format

C_INTEGER($urgency)
GET PICTURE METADATA(vPicture;IPTC urgency;$urgency)
```

## Example 3

Reception of tags with multiple values in arrays

```
ARRAY TEXT($tTkeywords;0)
GET PICTURE METADATA(vPicture;IPTC keywords;$tTkeywords)
```

After execution of the command, arrTkeywords contains for example:

```
$arrTkeywords{1}="France"
$arrTkeywords{2}="Europe"
```

## Example 4

Reception of tags with multiple values in a Text variable

```
C_TEXT($vTwords;0)
GET PICTURE METADATA(vPicture;IPTC keywords;$vTwords)
```

After execution of the command, *vTwords* contains for example "France;Europe".

## System variables and sets

The *OK* system variable returns 1 if the retrieval of the metadata has proceeded correctly and 0 if an error occurs or if at least one of the tags is not found. In all cases, the any values that can be read are returned.

# Is picture file

Is picture file ( filePath {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| filePath | Text | ⇒ | File pathname |
| * | Operator | ⇒ | Validate data |
| Function result | Boolean | ⇐ | True = filePath designates a picture file; otherwise, False |

## Description

The **Is picture file** command tests the file designated by the *filePath* parameter and returns True if it is a valid picture file. The command returns False if the file is not of the picture type or if it is not found.

Pass the pathname of the picture file to be tested in the *filePath* parameter. The path must be expressed with the system syntax. You can pass an absolute pathname or a pathname relative to the database structure file. If you pass an empty string (""), the command returns False.

If you do not pass the * parameter, the command tests the file by looking for its extension among the list of available codecs. If you want to be able to test files without extensions or to carry out a more thorough verification, pass the * parameter. In this case, the command makes additional tests: it loads and inspects the file header and queries the codecs in order to validate the picture. This syntax slows command execution.

**Note:** The command returns True for PDF files under Windows and EMF files under Mac OS.

# PICTURE CODEC LIST

PICTURE CODEC LIST ( codecArray {; namesArray}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| codecArray | String array | ⇦ | IDs of available picture Codecs |
| namesArray | String array | ⇦ | Names of picture Codecs |
| * | Operator | ⇨ | Return list of reading (decoding) Codecs |

## Description

The **PICTURE CODEC LIST** command fills the *codecArray* array with the list of picture Codec IDs that are available on the machine where it is executed. This list includes the Codec IDs of picture formats that are managed natively by 4D.

The Codec IDs can be returned in the *codecArray* array in the following forms:

- As an extension (for example, ".gif")
- As a Mime type (for example, "image/jpeg")

**Compatibility note:** If QuickTime has been enabled in the database (see the **Pictures** section), 4-character QuickTime codes can also be returned (for example "PNTG").

The form returned by the command will depend on the way the Codec is recorded at the operating system level. The optional *namesArray* array can be used to retrieve the name of each Codec. These names are more explicit than the IDs. This array can be used, for example, to build and display a menu listing the available Codecs.

By default, if you do not pass the * parameter, the command returns only the Codecs that can be used to encode (write) pictures. These IDs can be used in the *format* parameter of the picture export commands **WRITE PICTURE FILE** and **PICTURE TO BLOB**.

If you pass the * parameter, the command also returns the list of codecs used for decoding (reading) the pictures. The two lists are not exclusive, certain reading and writing Codecs are identical. Codecs intended for encoding pictures may usually be used for decoding. On the other hand, decoding Codecs cannot necessarily be used for encoding. For example, the ".jpg" Codec will be found in both lists, whereas the ".xbmp" Codec will only be found in the list of reading (decoding) Codecs.

# PICTURE LIBRARY LIST

PICTURE LIBRARY LIST ( picRefs ; picNames )

| Parameter | Type | | Description |
|---|---|---|---|
| picRefs | Longint array | ⇦ | Reference numbers of the Picture Library graphics |
| picNames | String array | ⇦ | Names of the Picture Library graphics |

## Description

The **PICTURE LIBRARY LIST** command returns the reference numbers and names of the pictures currently stored in the Picture Library of the database.

After the call, you retrieve the reference numbers in the array *picRefs* and the names in the array *picNames*. The two arrays are synchronized: the nth element of *picRefs* is the reference number of the Picture Library graphic whose name is returned in the nth element of *picNames*.

If necessary, the command automatically creates and sizes the *picRefs* and *picNames* arrays.

The maximum length of a Picture Library graphic name is 255 characters.

If there are no pictures in the Picture Library, both arrays are returned empty.

To obtain the number of pictures currently stored in the Picture Library, use the **Size of array** command to get the size of one of the two arrays.

## Example 1

The following code returns the catalog of the Picture Library in the arrays *alPicRef* and *asPicName*:

```
PICTURE LIBRARY LIST(alPicRef;asPicName)
```

## Example 2

The following example tests whether or not the Picture Library is empty:

```
PICTURE LIBRARY LIST(alPicRef;asPicName)
If(Size of array(alPicRef)=0)
   ALERT("The Picture Library is empty.")
Else
   ALERT("The Picture Library contains "+String(Size of array(alPicRef))+" pictures.")
End if
```

## Example 3

The following example exports the Picture Library to a document on disk:

```
PICTURE LIBRARY LIST($alPicRef;$asPicName)
$vlNbPictures:=Size of array($alPicRef)
If($vlNbPictures>0)
   SET CHANNEL(12;"")
   If(OK=1)
      $vsTag:="4DV6PICTURELIBRARYEXPORT"
      SEND VARIABLE($vsTag)
      SEND VARIABLE($vlNbPictures)
      gError:=0
      For($vlPicture;1;$vlNbPictures)
         $vlPicRef:=$alPicRef{$vlPicture}
         $vsPicName:=$asPicName{$vlPicture}
         GET PICTURE FROM LIBRARY($alPicRef{$vlPicture};$vgPicture)
         If(OK=1)
            SEND VARIABLE($vlPicRef)
```

```
            SEND VARIABLE($vsPicName)
            SEND VARIABLE($vgPicture)
         Else
            $vlPicture:=$vlPicture+1
            gError:=-108
         End if
      End for
      SET CHANNEL(11)
      If(gError#0)
         ALERT("The Picture Library could not be exported, retry with more memory.")
         DELETE DOCUMENT(Document)
      End if
   End if
Else
   ALERT("The Picture Library is empty.")
End if
```

# PICTURE PROPERTIES

PICTURE PROPERTIES ( picture ; width ; height {; hOffset {; vOffset {; mode}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇒ | Picture for which to get information |
| width | Longint | ⇐ | Width of the picture expressed in pixels |
| height | Longint | ⇐ | Height of the picture expressed in pixels |
| hOffset | Longint | ⇐ | Horizontal offset when displayed on background |
| vOffset | Longint | ⇐ | Vertical offset when displayed on background |
| mode | Longint | ⇐ | Transfer mode when displayed on background |

## Description

The **PICTURE PROPERTIES** command returns information about the picture you pass in *picture*.

The *width* and *height* parameters return the width and height of the picture.

The *hOffset*, *vOffset*, and *mode* parameters return the horizontal and vertical positions and the transfer mode of the picture when displayed on the background in a form (“On Background”).

# Picture size

Picture size ( picture ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇒ | Picture for which to return the size in bytes |
| Function result | Longint | ⇐ | Size in bytes of the picture |

## Description

**Picture size** returns the size of *picture* in bytes.

# PICTURE TO BLOB

PICTURE TO BLOB ( picture ; pictureBlob ; codec )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇒ | Picture field or variable |
| pictureBlob | BLOB | ⇐ | BLOB to receive the converted picture |
| codec | String | ⇒ | Picture Codec ID |

## Description

The **PICTURE TO BLOB** command converts a picture stored in a 4D variable or field to another format and places the resulting picture in a BLOB.

A picture 4D field or variable is passed in the *picture* parameter. In the *pictureBlob* parameter is passed a BLOB variable or field which should contain the converted picture.

Pass in the *codec* parameter a string setting the conversion format.

A Codec can be an extension (for example, ".gif") or a Mime type (for example "image/jpeg"). You can get a list of available Codecs via the **PICTURE CODEC LIST** command.

Once the command has been executed, the *pictureBlob* contains the picture in the specified format.

If the conversion was successful, the system variable OK is set to 1. If the conversion has failed (converter not available), OK is set to 0 and the generated BLOB is empty (0 byte).

# PICTURE TO GIF

PICTURE TO GIF ( pict ; blobGIF )

| Parameter | Type | | Description |
|---|---|---|---|
| pict | Picture | ⇒ | Picture field or picture variable |
| blobGIF | BLOB | ⇐ | BLOB containing the GIF picture |

## Description

The **PICTURE TO GIF** command converts a PICT picture stored in a variable or in a 4D field into a GIF picture.

You pass a picture variable or a picture field in *pict* and a BLOB variable or a BLOB field in *blobGIF*. After executing the command, *blobGIF* contains the image in GIF format.

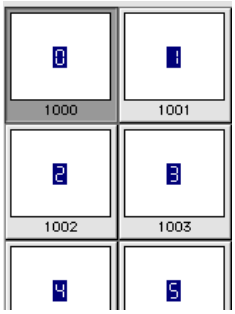
**Note:** The GIF picture format cannot contain more than 256 colors. If the original PICT picture contains more colors, some may be lost. The command reduces the number of colors according to the system palette. The GIF generated is of type 87a (opaque) and normal (not interlaced).

You can then save the picture located in *blobGIF* in a file using the **Windows Ctrl down** command or you can even publish it on the Web.

If the conversion was successful, the *OK* system variable is set to 1. Otherwise, it will be equal to 0.

## Example

Let us assume that you want to generate a GIF picture on the fly by displaying a connection counter. In the database's picture library, place all the numbers as pictures:




In the **On Web Connection Database Method**, you write the following code:

```
If(Web Context)
   ...
Else
   C_BLOB($blob)
   Case of
         ...
      :($1="/4dcgi/counter") `Generating a GIF counter
 `When 4D detects this URL while sending the static page
         $blob:=gifcounter(◊nbHits) `Calculates the GIF picture
 `The ◊nbHits variable contains the number of connections
         WEB SEND BLOB($blob;"image/gif")
 `Insert the picture and send it to the browser
         ...
   End case
End if
```

Here is the gifcounter method:

```
C_LONGINT($1)
C_PICTURE($img)
C_BLOB($0)
If($1=0)
   $ndigits:=1
Else
```

```
   $ndigits:=1+Length(String($1))
End if
If($ndigits<5)
   $ndigits:=5
End if

$div:=10^($ndigits-1)
For($i;1;$ndigits)
   $ref:=Int($1/$div)%10
   GET PICTURE FROM LIBRARY($ref+1000;picture)
   $img:=$img+picture
   $div:=$div/10
End for

PICTURE TO GIF($img;$0)
```

When sending a page to the Web browser, 4D displays a GIF picture that looks like the following picture:

## System variables and sets

If the conversion was successful, the *OK* system variable is set to 1. Otherwise, it will be equal to 0.

# READ PICTURE FILE

READ PICTURE FILE ( fileName ; picture {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| fileName | String | ⇒ | Name or full pathname of the file to read, or empty string |
| picture | Picture | ⇐ | Field or variable receiving picture |
| * | Operator | ⇒ | If passed = accept any type of file |

## Description

The **READ PICTURE FILE** command opens the picture saved in the *fileName* disk file and loads it in the *picture* 4D field or variable.

You can pass in *fileName* the full pathname of the file to read, or a file name only. If you pass only the file name, it should be located next to the database structure file. Under Windows, the file extension must be indicated.
If an empty string ("") is passed in *fileName*, the standard Open file dialog box appears and the user selects the file to be read, as well as the available formats.

You can get the list of available formats using the **PICTURE CODEC LIST** command.

You pass in *picture* the picture variable or field which will receive the picture read.

**Note:** The internal picture format is stored within the 4D variable or field.

If you pass the optional * parameter, the command will accept any type of file. This means that you can work with pictures without necessarily having the suitable codecs (see the description of the **BLOB TO PICTURE** command).

## System variables and sets

If the command is executed successfully, the system variable Document contains the full pathname to the open file and the system variable OK is set to 1. Otherwise, OK is set to 0.

# REMOVE PICTURE FROM LIBRARY

REMOVE PICTURE FROM LIBRARY ( picRef | picName )

| Parameter | Type | | Description |
|---|---|---|---|
| picRef \| picName | Longint, String | ⇒ | Reference number of Picture Library graphic or Name of Picture Library graphic |

## Description

The **REMOVE PICTURE FROM LIBRARY** command removes from the Picture Library the picture whose reference number is passed in *picRef* or whose name is passed in *picName*.

If there is no picture with that reference number or name, the command does nothing.

**4D Server: REMOVE PICTURE FROM LIBRARY** cannot be used from within a method executed on the server machine (stored procedure or trigger). If you call **REMOVE PICTURE FROM LIBRARY** on a server machine, nothing happens—the call is ignored.

**Warning:** Design objects (hierarchical list items, menu items, etc.) may refer to Picture Library graphics. Use caution when deleting a Picture Library graphic programmatically.

## Example 1

The following example deletes the picture #4444 from the Picture Library.

```
REMOVE PICTURE FROM LIBRARY(4444)
```

## Example 2

The following example deletes from the Picture Library any pictures whose names begin with a dollar sign (*$*):

```
PICTURE LIBRARY LIST($alPicRef;$asPicName)
For($vlPicture;1;Size of array($alPicRef))
   If($asPicName{$vlPicture}="$@")
      REMOVE PICTURE FROM LIBRARY($alPicRef{$vlPicture})
   End if
End for
```

# SET PICTURE FILE NAME

SET PICTURE FILE NAME ( picture ; fileName )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture field, Picture variable | ⇒ | Picture for which to set the default name |
| fileName | Text | ⇒ | Default picture name |

## Description

The **SET PICTURE FILE NAME** command sets or changes the default file name for the picture passed as parameter.

This name may have been set automatically based on the original name of the picture file imported into the picture field or variable or during a prior call to **SET PICTURE FILE NAME**.

The default name is used as the file name when the picture is exported in a disk file. If the contents of the field are copied into a varaible or into another field, the default name is also copied. For more information, refer to the *Design Reference* manual.

# SET PICTURE METADATA

SET PICTURE METADATA ( picture ; metaName ; metaContents {; metaName2 ; metaContents2 ; ... ; metaNameN ; metaContentsN} )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇒ | Picture whose metadata you want to set |
| metaName | Text | ⇒ | Name or path of block to set |
| metaContents | Variable | ⇒ | Metadata contents |

## Description

The **SET PICTURE METADATA** command lets you set or modify the contents of the metadata (or meta-tags) found in the *picture* (4D picture field or variable), when they are modifiable.

Metadata are additional information inserted into pictures. 4D lets you handled four types of standard metadata: EXIF, GPS, IPTC and TIFF.

**Note:** For a detailed description of these metadata types, you can consult the following documents: http://www.iptc.org/std/IIM/4.1/specification/IIMV4.1.pdf (IPTC) and http://exif.org/Exif2-2.PDF (TIFF, EXIF and GPS).

In the *metaName* parameter, pass a string specifying the type of metadata to set or modify. You can pass:

- one of the constants from the **Picture Metadata Names** theme. This theme groups together all the tags supported by 4D. Each constant contains a tag path (for example, "TIFF/DateTime"),
- the name of a complete block of metadata ("TIFF", "EXIF", "GPS" or "IPTC"),
- an empty string ("").

Pass the new values of the metadata in the *metaContents* parameter:

- If you passed a tag path constant in *metaName*, in the contents parameter you can pass the value to set directly or one of the appropriate constants from the **Picture Metadata Values** theme. The value can be of the Text, Longint, Real, Date or Time type, according to the metadata specified. You can use an array if the metadata contains more than one value. If you pass a string, it must be formated in XML (XMP standard). You can pass an empty string ("") in order to erase any existing metadata.
- If you passed a block name or an empty string in *metaName*, in the *metaContents* parameter you can pass the XML DOM reference of the element containing the metadata to set. In the case of an empty string, all the metadata will be modified.

**Warning:** Certain metadata are read only and therefore cannot be modified by the **SET PICTURE METADATA** command, for example TIFF XResolution/TIFF YResolution, EXIF Color Space or EXIF Pixel X Dimension/EXIF Pixel Y Dimension.

Under Windows, if an error occurs during execution of the command, the *OK* variable is set to 0. Note that under Mac OS, for technical reasons, metadata writing errors are not detected. Therefore this command does not modify the *OK* variable under Mac OS.

**Notes:**

- Only certain picture formats (more specifically, JPEG and TIFF) support metadata. Conversely, formats such as GIF or BMP do not accept metadata. When you convert a picture with metadata to a format that does not support it, then information is lost.
- Under OS X version 10.7 (Lion), a bug in the native framework used for encoding and decoding picture metadata may cause inaccuracies in GPS coordinates. In this case, updating to OS X 10.8 (Mountain Lion) or 10.9 (Maverick) is strongly recommended.

## Example 1

Setting several values of the "Keywords" metadata via arrays:

```
ARRAY TEXT($arrTkeywords;2)
$arrTkeywords{1}:="France"
$arrTkeywords{2}:="Europe"
SET PICTURE METADATA(vPicture;IPTC keywords;$arrTkeywords)
```

## Example 2

Setting of GPS block via a DOM reference:

```
C_TEXT($domMetas)
$domMetas:=DOM Parse XML source("metas.xml")
C_TEXT($gpsRef)
$gpsRef:=DOM Find XML element($domMetas;"Metadatas/GPS")
If(OK=1)
   SET PICTURE METADATA(vImage;"GPS";$refGPS)
 //here $gpsRef actually points to the GPS element
   ...
End if
DOM CLOSE XML($domMetas)
```

## Note

When all the metadata are handled via a DOM element reference, the tags are stored as attributes attached to an element (a child of the referenced element) whose name is the block name (TIFF, IPTC, etc.). When a specific metadata block is manipulated, the block tags are stored as attributes that are directly attached to the element referenced by the command.

# SET PICTURE TO LIBRARY

SET PICTURE TO LIBRARY ( picture ; picRef ; picName )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ➡ | New picture |
| picRef | Longint | ➡ | Reference number of Picture Library graphic |
| picName | String | ➡ | New name of the picture |

## Description

The **SET PICTURE TO LIBRARY** command creates a new picture or replaces a picture in the Picture Library.

Before the call, you pass:

- the picture reference number in *picRef* (range 1...32767 )
- the picture itself in *picture*.
- the name of the picture in *picName* (maximum length: 255 characters).

If there is an existing Picture Library graphic with the same reference number, the picture contents are replaced and the picture is renamed according to the values passed in *picture* and *picName.*

If there is no Picture Library graphic with the reference number passed in *picRef*, a new picture is added to the Picture Library.

**4D Server: SET PICTURE TO LIBRARY** cannot be used from within a method executed on the server machine (stored procedure or trigger). If you call **SET PICTURE TO LIBRARY** on a server machine, nothing happens—the call is ignored.

**Warning:** Design objects (hierarchical list items, menu items, etc.) may refer to Picture Library graphics. Use caution when modifying a Picture Library graphic programmatically.

**Note:** If you pass an empty picture in *picture* or a negative or null value in *picRef*, the command does nothing.

## Example 1

No matter what the current contents of the Picture Library, the following example adds a new picture to the Picture Library by first looking for a unique picture reference number:

```
PICTURE LIBRARY LIST($alPicRef;$asPicNames)
Repeat
   $vlPicRef:=1+Abs(Random)
Until(Find in array($alPicRef;$vlPicRef)<0)
SET PICTURE TO LIBRARY(vgPicture;$vlPicRef;"New Picture")
```

## Example 2

The following example imports into the Picture Library the pictures (stored in a document on disk) created by the third example for the command **PICTURE LIBRARY LIST**:

```
SET CHANNEL(10;"")
If(OK=1)
   RECEIVE VARIABLE($vsTag)
   If($vsTag="4DV6PICTURELIBRARYEXPORT")
      RECEIVE VARIABLE($vlNbPictures)
      If($vlNbPictures>0)
         For($vlPicture;1;$vlNbPictures)
            RECEIVE VARIABLE($vlPicRef)
            If(OK=1)
               RECEIVE VARIABLE($vsPicName)
            End if
            If(OK=1)
               RECEIVE VARIABLE($vgPicture)
            End if
```

```
            If(OK=1)
               SET PICTURE TO LIBRARY($vgPicture;$vlPicRef;$vsPicName)
            Else
               $vlPicture:=$vlNbPictures+1
               ALERT("This file looks like being damaged.")
            End if
         End for
      Else
         ALERT("This file looks like being damaged.")
      End if
   Else
      ALERT("The file  ""+Document+""  is not a Picture Library export file.")
   End if
   SET CHANNEL(11)
   End
```

## Error management

If there is not enough memory to add the picture to the Picture Library, an error -108 is generated. Note that I/O errors may also be returned (i.e., the structure file is locked). You can catch these errors using an error-handling method.

# TRANSFORM PICTURE

TRANSFORM PICTURE ( picture ; operator {; param1 {; param2 {; param3 {; param4}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇒ | Source picture to be transformed |
| | | ⇐ | Resulting picture after transformation |
| operator | Longint | ⇒ | Type of transformation to be done |
| param1 | Real | ⇒ | Transformation parameter |
| param2 | Real | ⇒ | Transformation parameter |
| param3 | Real | ⇒ | Transformation parameter |
| param4 | Real | ⇒ | Transformation parameter |

## Description

The **TRANSFORM PICTURE** command applies a transformation of the *operator* type to the picture passed in the *picture* parameter.

**Note:** This command extends the functionalities offered by conventional picture transformation operators (+/, etc., see the **Picture Operators** section*)*. These operators remain entirely usable in 4D.

The source *picture* is modified directly after execution of the command. Note that certain operations are not destructive and can be reversed by performing the opposite operation or by means of the “Reset” operation. For example, a picture reduced to 1% will regain its original size with no alteration if it is enlarged by a factor of 100 subsequently. Transformations do not modify the original picture type: for example, a vectorial picture will remain vectorial after its transformation.

In *operator*, pass the number of the operation to be carried out and in *param*, the parameter(s) needed for this operation (the number of parameters depends on the operation). You can use one of the constants of the "**Picture Transformation**" theme in *operator*. These operators and their parameters are described in the following table:

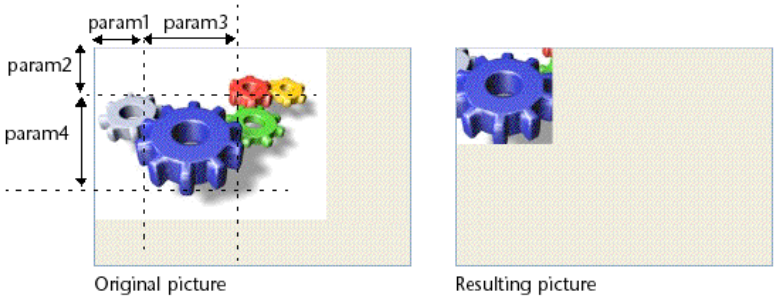
| operator (value) | param1 | param2 | param3 | param4 | Values | Cancellable |
|---|---|---|---|---|---|---|
| Reset (0) | - | - | - | - | - | - |
| Scale (1) | Width | Height | - | - | Factors | Yes |
| Translate (2) | X axis | Y axis | - | - | Pixels | Yes |
| Flip horizontally (3) | - | - | - | - | | Yes |
| Flip vertically (4) | - | - | - | - | | Yes |
| Crop (100) | X Orig. | Y Orig. | Width | Height | Pixels | No |
| Fade to grey scale (101) | - | - | - | - | | No |
| Transparency (102) | RGB color | - | - | - | Hexadecimal | No |

- Reset: All matrix operations carried out on the picture (scale, flip, and so on) are undone.
- Scale: The picture is resized horizontally and vertically according to the values passed respectively in *param1* and *param2*. These values are factors: for example, to enlarge the width by 50%, pass 1.5 in *param1* and to reduce the height by 50%, pass 0.5 in *param2*.
- Translate: The picture is moved by *param1* pixels horizontally and by *param2* pixels vertically. Pass a positive value to move to the right or towards the bottom and a negative value to move to the left or towards the top.
- Flip horizontally and Flip vertically: The original picture is flipped. Any movement that was carried out beforehand will not be taken into account.
- Crop: The picture is cropped starting from the point of the *param1* and *param2* coordinates (expressed in pixels). The width and height of the new picture is determined by the *param3* and *param4* parameters. This transformation cannot be undone.
- Fade to grey scale: The picture is switched to gray scale (no parameter is required). This transformation cannot be undone.
- Transparency: A transparency mask is applied to the picture based on the color passed in *param1*. For example, if you pass 0x00FFFFFF (white color) in *param1*, all the white pixels in the original picture will be transparent in the transformed picture. This operation can be applied to bitmap or vector pictures. By default, if the *param1* parameter is omitted, the color white (0x00FFFFFF) is set as the target color. This function is specially designed to handle transparency in pictures converted from the deprecated PICT format pictures, but can be used with pictures of any type. This transformation cannot be undone.

## Example 1

Here is an example of cropping a picture (the picture is displayed in the form with the “Truncated (non-centered)” format):

```
TRANSFORM PICTURE($vpGears;Crop;50;50;100;100)
```

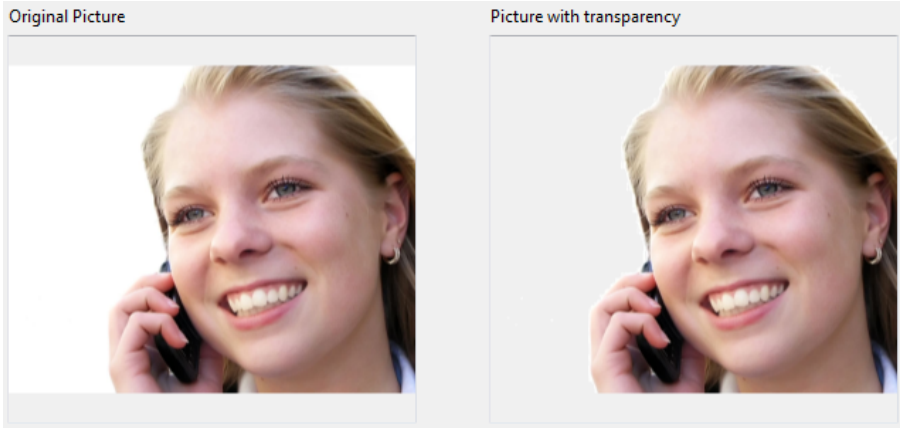



## Example 2

You want to set the white parts of a picture to transparent. To do this, you can use the following code:

```
TRANSFORM PICTURE(Pict1;Transparency;0x00FFFFFF) //0x00FFFFFF is white
```

You will get the following result:



# WRITE PICTURE FILE

WRITE PICTURE FILE ( fileName ; picture {; codec} )

| Parameter | Type | | Description |
|---|---|---|---|
| fileName | Alpha | ⇒ | Name or full pathname of the file to write, or empty string |
| picture | Picture | ⇒ | Picture field or variable to write |
| codec | String | ⇒ | Picture Codec ID |

## Description

The **WRITE PICTURE FILE** command saves the picture passed in the *picture* parameter in the defined *codec* to disk.

You can pass in *fileName* the full pathname to the file to create, or a file name only. If you just pass the file name, the file will be located next to the database structure file. The file extension has to be indicated.

If an empty string ("") is passed in *fileName*, the standard Save file dialog box is displayed and the user can indicate the name, location and format of the file to create. If a default name is associated with the Picture field, it is provided in the dialog box (see the **SET PICTURE FILE NAME** command).

You will pass in *picture* the picture variable or field which contains the picture to save on disk.

The optional *codec* parameter can be used to define the format in which the picture will be saved. A Codec can be an extension (for example, ".gif") or a Mime type (for example "image/jpeg"). You can get a list of available Codecs via the **PICTURE CODEC LIST** command.

If you omit the *codec* parameter, the command will attempt to determine the codec based on the extension of the file name passed in the *fileName* parameter. For example, if you pass the statement:

```
WRITE PICTURE FILE("c:¥folder¥photo.jpg";myphoto)
```

... the command will use the JPEG codec to store the picture.
If the extension used does not correspond to any available codec, the file is not saved and the OK system variable is set to 0. If you do not pass a *codec* or a file extension, the picture file is saved in PICT format.

**Note:** If the write format of the *picture* (indicated via the extension of *fileName* or the *codec* parameter) is the same the as its original type and if no transformation operation has been applied to it, the picture is written "as is", without any modification.

If the command is executed successfully, the system variable Document contains the full pathname to the file created and the system variable OK is set to 1. Otherwise, OK is set to 0.

# _o_PICTURE TYPE LIST

_o_PICTURE TYPE LIST ( formatArray {; nameArray} )

| Parameter | Type | | Description |
|---|---|---|---|
| formatArray | String array | ⇦ | QuickTime codes for the available import/export formats |
| nameArray | String array | ⇦ | Format names |

## Compatibility Note:

This command requires QuickTime and does not provide access to formats managed natively by 4D since version 11. It is thus of limited interest and can be replaced favorably by the **PICTURE CODEC LIST** command.

# _o_QT COMPRESS PICTURE

_o_QT COMPRESS PICTURE ( picture ; method ; quality )

| Parameter | Type | | Description |
|---|---|---|---|
| picture | Picture | ⇒ | Picture to be compressed |
| | | ⇐ | Compressed picture |
| method | String | ⇒ | 4-character string compression method |
| quality | Longint | ⇒ | Compression quality (1..1000) |

## Compatibility note

This command calls for obsolete mechanisms and must be replaced by the **CONVERT PICTURE** command.

# _o_QT COMPRESS PICTURE FILE

_o_QT COMPRESS PICTURE FILE ( document ; method ; quality )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Document reference number |
| method | String | ⇒ | 4-character string compression method |
| quality | Longint | ⇒ | Compression quality (1..1000) |

## Compatibility note

This command calls for obsolete mechanisms and must now be replaced by the **WRITE PICTURE FILE** or **PICTURE TO BLOB** commands.

# _o_QT LOAD COMPRESS PICTURE FROM FILE

_o_QT LOAD COMPRESS PICTURE FROM FILE ( document ; method ; quality ; picture )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Document reference number |
| method | String | ⇒ | 4-character string compression method |
| quality | Longint | ⇒ | Compression quality (1..1000) |
| picture | Picture | ⇐ | Compressed picture |

## Compatibility note

This command calls for obsolete mechanisms and must now be replaced by the **READ PICTURE FILE** and **CONVERT PICTURE** commands.

# _o_SAVE PICTURE TO FILE

_o_SAVE PICTURE TO FILE ( document ; picture )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Document reference number |
| picture | Picture | ⇒ | Picture to be saved |

## Compatibility note

This command calls for obsolete mechanisms and is only kept for compatibility reasons. It has been favorably replaced by the **WRITE PICTURE FILE** command.

# Printing

- Integration of PDFCreator driver under Windows
- ACCUMULATE
- BLOB to print settings New 16.0
- BREAK LEVEL
- CLOSE PRINTING JOB
- Get current printer
- Get print marker
- GET PRINT OPTION Updated 16.0
- Get print preview
- GET PRINTABLE AREA
- GET PRINTABLE MARGIN
- Get printed height
- Is in print preview
- Level
- OPEN PRINTING JOB
- PAGE BREAK
- PAGE SETUP
- Print form
- PRINT LABEL
- Print object
- PRINT OPTION VALUES
- PRINT RECORD
- PRINT SELECTION
- PRINT SETTINGS
- Print settings to BLOB New 16.0
- PRINTERS LIST Updated 16.0
- Printing page
- SET CURRENT PRINTER
- SET PRINT MARKER
- SET PRINT OPTION Updated 16.0
- SET PRINT PREVIEW
- SET PRINTABLE MARGIN
- Subtotal

# Integration of PDFCreator driver under Windows

Support for PDF printing differs depending on the version of Windows:

- for Windows 8 and previous versions, you need to use the PDFCreator driver.
- starting with Windows 10, a native Microsoft driver is integrated.

**Note:** Under Mac OS, PDF printing is supported natively by the system.

## Windows 8 and previous versions

The support of PDF printing under Windows relies on the PDFCreator driver to offer simple and functional PDF printing functions. The **GET PRINT OPTION** and **SET PRINT OPTION** commands both make use of this driver.

PDFCreator is a free driver (OpenSource) governed by the AFPL (Aladdin Free Public License). To use the PDFCreator driver, you must download the appropriate version and install it in your environment. (It is not installed by default by 4D.) You must have Administrator access rights in order to install the driver. You can download PDFCreator here:
http://sourceforge.net/projects/pdfcreator/files/PDFCreator/

**Warning:** you must use a PDFCreator version that is **compatible with 4D**. To find out the compatible and certified versions of PDFCreator, please consult the certification matrices of 4D products, found on the **Resources page** (**Compatibility section**) of the 4D Web site.

During installation, a new virtual printer named "PDFCreator" by default is installed in your system. You can change this name if desired.

## Starting with Windows 10

Windows 10 includes a native PDF driver which allows 4D to create PDFs directly without needing to use a third-party driver like PDFCreator.

The driver name is "Microsoft Print to PDF".

Here is an example of creating a PDF document under Windows 10 using 4D print commands:

```
$pdfpath:=System folder(Desktop)+"test.pdf"

$pdfprintername:="Microsoft Print to PDF"
ARRAY TEXT($name1;0)
PRINTERS LIST($name1)
If(Find in array($name1;$pdfprintername)>0)
   SET CURRENT PRINTER($pdfprintername)
   SET PRINT OPTION(Destination option;2;$pdfpath)
   ALL RECORDS([Table_1])
   PRINT SELECTION([Table_1];*)
   SET CURRENT PRINTER("")
End if
```

# ACCUMULATE

ACCUMULATE ( data {; data2 ; ... ; dataN} )

| Parameter | Type | | Description |
|---|---|---|---|
| data | Field, Variable | ⇨ | Numeric field or variable on which to accumulate |

## Description

**ACCUMULATE** specifies the fields or variables to be accumulated during a form report performed using **PRINT SELECTION**.

You **must** execute **BREAK LEVEL** and **ACCUMULATE** before every report for which you want to do break processing. These commands activate break processing for a report. See the explanation for the **Subtotal** command.

Use **ACCUMULATE** when you want to include subtotals for numeric fields or variables in a form report. **ACCUMULATE** tells 4D to store subtotals for each of the Data arguments. The subtotals are accumulated for each break level specified with the **Subtotal** command.

Execute **ACCUMULATE** before printing the report with **PRINT SELECTION**.

Use the **Subtotal** function in the form method or an object method to return the subtotal of one of the *data* arguments.

## Example

See the example for the **BREAK LEVEL** command.

# BLOB to print settings

BLOB to print settings ( printSettings {; params} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| printSettings | BLOB | ⇒ | BLOB containing print settings |
| params | Longint | ⇒ | 0=Restore saved values for number of copies and page range, 1=Reset to default values |
| Function result | Longint | ⇐ | Status code: 1=Operation successful, 0=No current printer, -1=Incorrect parameters, 2=Printer changed |

## Description

The **BLOB to print settings** command replaces the current 4D print settings with the parameters stored in the *printSettings* BLOB. This BLOB must have been generated by the **Print settings to BLOB** command or the **_o_AP Print settings to BLOB** 4D Pack command (see below).

The *params* parameter allows you to define how to handle the basic "number of copies" and "page range" settings:

- If you pass 0 or omit this parameter, the values stored in the BLOB are restored,
- If you pass 1, the values are reset to default: the number of copies is set to 1 and the page range is set to "all pages".

The print settings are applied to the current printer and for the whole session, as long as no command such as **PAGE SETUP**, **SET PRINT OPTION** or **PRINT SELECTION** without the > parameter modifies them. The parameters set are used more particularly by the **PRINT SELECTION**, **PRINT LABEL**, **PRINT RECORD**, **Print form** and **QR REPORT** commands, as well as by the menu commands of 4D, including those of the Design environment.

The **PRINT SELECTION**, **PRINT LABEL**, and **PRINT RECORD** commands must be called with the > parameter (if applicable) in order for the settings defined by **BLOB to print settings** to be kept.

The command returns one of the following status codes:

- -1: the BLOB is incorrect,
- 0: no current printer is selected (in this case the command does nothing),
- 1: the BLOB has been correctly loaded,
- 2: the BLOB has been correctly loaded but the current printer name has changed(*).
  **Note:** Code (2) is always returned if the BLOB was created by the **_o_AP Print settings to BLOB** 4D Pack command, even if the printer name did not actually change, since this information was not included in the 4D Pack BLOBs.

(*) Settings depend on the currently selected printer at the moment the BLOB was saved. Applying these settings to a another printer is supported if both printers are of the same model. If the printers are different, only common parameters will be restored.

### Windows / OS X

The *printSettings* BLOB can be saved and read on both platforms. However, even if some print settings are common, some others are platform-specific and depend on the drivers and system versions. If the same *printSettings* BLOB is shared between both platforms, you may lose parts of the information.
When used in a heterogeneous environment, in order to restore the maximum settings available for each platform (and not only the common part), it is recommended that you work with two *printSettings* BLOBs, one for each platform.

### Compatibility with 4D Pack commands

The print settings BLOBs generated by the legacy **_o_AP Print settings to BLOB** command from 4D Pack can be loaded and used by the **BLOB to print settings** command. Note however that if they are saved using **Print settings to BLOB**, they are converted and can no longer be opened using **_o_AP BLOB to print settings**. The **BLOB to print settings** command stores much more printing information than **_o_AP Print settings to BLOB**.

## Example

You want to apply print settings previously saved to disk to the current 4D printing context:

```
C_BLOB(curSettings)
DOCUMENT TO BLOB(Get 4D folder(Active 4D Folder)+"current4Dsettings.blob";curSettings)
 //current4Dsettings has been created by Print settings to BLOB
$err:=BLOB to print settings(curSettings;0)
```

```
Case of
   :($err=1)
 //everything is OK
   :($err=2)
      CONFIRM("Printer has changed!¥n¥nCheck the print settings?")
      If(OK=1)
         PRINT SETTINGS
      End if
   :($err=0)
      ALERT("There is no current printer.")
   :($err=-1)
      ALERT("Invalid settings file.")
End case
```

# BREAK LEVEL

BREAK LEVEL ( level {; pageBreak} )

| Parameter | Type | | Description |
|---|---|---|---|
| level | Longint | ⇒ | Number of break levels |
| pageBreak | Longint | ⇒ | Break level for which to do a page break |

## Description

**BREAK LEVEL** specifies the number of break levels in a report performed using **PRINT SELECTION**.

You **must** execute **BREAK LEVEL** and **ACCUMULATE** before every report for which you want to do break processing. These commands activate break processing for a report. See the explanation for the **Subtotal** command.

The *level* parameter indicates the deepest level for which you want to perform break processing. You must have sorted the records with at least that many levels. If you have sorted more levels, those levels will be printed as sorted, but will not be processed for breaks.

Each break level that is generated will print the corresponding Break areas and Header areas in the form. There should be at least as many Break areas in the form as the number you pass in *level*. If there are more Break areas, they will be ignored and will not be printed.

The second, optional, argument, *pageBreak*, is used to cause page breaks during printing.

## Example

The following example prints a report with two break levels. The selection is sorted on four levels, but the **BREAK LEVEL** command specifies to break on only two levels. One field is accumulated with the **ACCUMULATE** command:

```
ORDER BY([Emp]Dept;>;[Emp]Title;>;[Emp]Last;>;[Emp]First;>) ` Sort on four levels
BREAK LEVEL(2) ` Turn on break processing to 2 levels (Dept and Title)
ACCUMULATE([Emp]Salary) ` Accumulate the salaries
FORM SET OUTPUT([Emp];"Dept salary") ` Select the report form
PRINT SELECTION([Emp]) ` Print the report
```

# CLOSE PRINTING JOB

CLOSE PRINTING JOB
Does not require any parameters

## Description

The **CLOSE PRINTING JOB** command closes the print job previously opened by the **OPEN PRINTING JOB** command and sends any printing document that has been assembled to the current printer.

Once this command is executed, the printer again becomes available for other print jobs.

# Get current printer

Get current printer -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String |  | Name of the current printer |

## Description

The **Get current printer** command returns the name of the current printer defined in the 4D application. By default, on start-up of 4D, the current printer is the printer defined in the system.

If the current printer is managed using a print server (spooler), the complete access path (under Windows) or the name of the spooler (under Mac OS) is returned.

To obtain the list of available printers as well as additional information, use the **PRINTERS LIST** command. To modify the current printer, use the **SET CURRENT PRINTER** command.

**Note:** When the Generic PDF driver constant is used with **SET CURRENT PRINTER**, **Get current printer** returns "_4d_pdf_printer" or the actual name of the PDF driver, if applicable (option not available with 4D 32-bit).

## System variables and sets

If no printer is installed, the system variable OK is set to 0. Otherwise, it is set to 1.

# Get print marker

Get print marker ( markNum ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| markNum | Longint | ⇒ | Marker number |
| Function result | Longint | ⇐ | Position of the marker |

## Description

The **Get print marker** command enables you to get the current position of a marker during printing.

This command can be used in two contexts:

- During the On Header form event, in the context of **PRINT SELECTION** and **PRINT RECORD** commands.
- During the On Printing Detail form event, in the context of the **Print form** command.

The coordinates are returned in pixels (1 pixel = 1/72 inch).

Pass one of the constants of the **Form Area** theme in the *markNum* parameter:

| Constant | Type | Value |
|---|---|---|
| Form break0 | Longint | 300 |
| Form break1 | Longint | 301 |
| Form break2 | Longint | 302 |
| Form break3 | Longint | 303 |
| Form break4 | Longint | 304 |
| Form break5 | Longint | 305 |
| Form break6 | Longint | 306 |
| Form break7 | Longint | 307 |
| Form break8 | Longint | 308 |
| Form break9 | Longint | 309 |
| Form detail | Longint | 0 |
| Form footer | Longint | 100 |
| Form header | Longint | 200 |
| Form header1 | Longint | 201 |
| Form header10 | Longint | 210 |
| Form header2 | Longint | 202 |
| Form header3 | Longint | 203 |
| Form header4 | Longint | 204 |
| Form header5 | Longint | 205 |
| Form header6 | Longint | 206 |
| Form header7 | Longint | 207 |
| Form header8 | Longint | 208 |
| Form header9 | Longint | 209 |

## Example

Refer to the example of the **SET PRINT MARKER** command.

# GET PRINT OPTION

GET PRINT OPTION ( option ; value1 {; value2} )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ⇨ | Option number or PDF option code |
| value1 | Longint, Text | ⇦ | Value 1 of the option |
| value2 | Longint, Text | ⇦ | Value 2 of the option |

## Description

The **GET PRINT OPTION** command returns the current value(s) of a print option.

The *option* parameter enables you to specify the option to get. You can either get a standard option (longint), or a PDF option code (string). The command returns, in the *value1* and (optionally) *value2* parameters, the current value(s) of the specified *option*.

To specify a standard printing option, you can use of the following predefined constants, located in the "**Print Options**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Paper option | Longint | 1 | If you use only *value1*, it contains the name of the paper. If you use both parameters, *value1* contains the paper width and *value2* contains the paper height. The width and height are expressed in screen pixels. Use the **PRINT OPTION VALUES** command to get the name, height and width of all the paper formats offered by the printer. |
| Orientation option | Longint | 2 | *value1* only: 1=Portrait, 2=Landscape. If a different orientation option is used, **GET PRINT OPTION** returns 0 in *value1*.<br>**64-bit versions**: This option can be called within a print job, which means that you can switch from portrait to landscape, or vice versa, during the same print job. |
| Scale option | Longint | 3 | *value1* only: scale value in percentage. Be careful, some printers do not allow you to modify the scale. If you pass an invalid value, the property is reset to 100% at the time of printing. |
| Number of copies option | Longint | 4 | *value1* only: number of copies to be printed. |
| Paper source option | Longint | 5 | (Windows only) *value1* only: number corresponding to the index, in the array of trays returned by the **PRINT OPTION VALUES** command, of the paper tray to be used. This option can only be used under Windows. |
| Color option | Longint | 8 | (Windows only) *value1* only: code specifying the mode for handling color: 1=Black and white (monochrome), 2=Color.<br>**64-bit versions**: This option is not supported in 4D 64-bit versions (obsolete) |
| Destination option | Longint | 9 | *value1*: code specifying the type of print destination: 1=Printer, 2=(PC)/PS File (Mac), 3=PDF file, 5=Screen (OS X driver option).<br>If *value1* is different from 1 or 5, *value2* contains pathname for resulting document. This path will be used until another path is specified. If a file with the same name already exists at the destination location, it will be replaced. With **GET PRINT OPTION**, if the current value is not in the predefined list, *value1* contains -1 and the system variable OK is set to 1. If an error occurs, *value1* and the system variable OK are set to 0.<br>**Note**: Under Windows, you can set the printing destination to 3 (PDF File) when the PDF Creator driver has been installed. When the *(9;3;path)* values are passed, 4D automatically starts a "silent" PDF printing which takes into account any option codes that are passed (note that if you pass an empty string in *value2* or omit this parameter, a file saving dialog appears at the time of printing.) After printing, the current settings are restored. This simplifies control of printing PDFs for 4D and lets you write multi-platform code. When the *(9;3;path)* values are not passed, printing is not controlled by 4D and any PDF Creator option codes that were passed are ignored. |
| Double sided option | Longint | 11 | (Windows only) *value1*: 0=Single-sided or standard, 1=Double-sided. If *value1*=1, *value2* contains the binding: 0=Left binding (default value), 1=Top binding.<br>**Note:** This option can only be used under Windows. |
| Spooler document name option | Longint | 12 | *value1* only: name of the current print document, which appears in the list of spooler documents. The name defined by this statement will be used for all the print documents of the session for as long as a new name or an empty string is not passed. To use or restore standard operation (using the method name in the case of a method, the table name for a record, etc.), pass an empty string in *value1*. |
| Mac spool file format option | Longint | 13 | (Mac only) *value1* only: 0=print job in PDF mode (default value) 1=print job in PostScript mode.<br>**Notes:**<br>- This option has no effect under Windows.<br>- Under OS X, printing is done as a PDF by default. However, the PDF print driver does not support PICT pictures with encapsulated PostScript information — these pictures are generated, more particularly, by vectorial drawing software. To avoid this problem, this option lets you modify the print mode to use under OS X for the current session. Keep in mind that printing in PostScript mode can lead to undesired side effects.<br>**64-bit versions:** This option is not supported; it is replaced by the Generic PDF driver option of the **SET CURRENT PRINTER** command. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Hide printing progress option | Longint | 14 | *value1* only: 1=hide progress windows, 0=display progress windows (default). This option is particularly useful in the case of PDF printing under OS X.<br>**Note:** There is already a Printing progress option found in the Database Settings dialog box (Interface page). However, it is applied globally to the application and does not hide all the windows under OS X. |
| Page range option | Longint | 15 | *value1*=first page to print (default value is 1) and (optional) *value2*=number of the last page to print (default value -1 = end of document). |
| Legacy printing layer option | Longint | 16 | (4D 64-bit versions for Windows only) *value1* only: 1=select the GDI-based legacy printing layer for the subsequent printing jobs. 0=select the D2D printing layer (default).<br>**64-bit versions:** This selector is only supported on 4D 64-bit single-user applications on Windows; it is ignored on other platforms. It is mainly intended to allow legacy plug-ins to print inside 4D jobs in 4D 64-bit applications. |

A PDF option code consists of two parts, *OptionType* and *OptionName*, combined together as "*OptionType:OptionName*". For more information on PDF option codes and possible values, refer to the description of the **SET PRINT OPTION** command.

**Note:** The **GET PRINT OPTION** command mainly supports PostScript printers. You can use this command with other types of printers, such as PCL or Ink, but in this case, it is possible that some options may not be available.

## System variables and sets

The system variable OK is set to 1 if the command has been executed correctly; otherwise, it is set to 0.

# Get print preview

Get print preview -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean |  | True = Print preview,<br>False = No print preview |

## Description

The **Get print preview** command returns True if the **SET PRINT PREVIEW** command was called with the **True** value in the current process.

Note that the user can modify this option before validating the dialog box. To get the final printing mode, you must use the **Is in print preview** command.

# GET PRINTABLE AREA

GET PRINTABLE AREA ( height {; width} )

| Parameter | Type | | Description |
|---|---|---|---|
| height | Longint | ⇐ | Height of printable area |
| width | Longint | ⇐ | Width of printable area |

## Description

The **GET PRINTABLE AREA** command returns the size, in pixels, of the *height* and *width* parameters of the printable area. This size depends on the current printing parameters, the paper orientation, etc.

The sizes returned do not vary from one page to another (after a page break, for instance).

Associated with the **Get printed height** command, this command is useful for knowing the number of pixels available for printing or for centering an object on the page.

**Note:** For more information regarding Printing management and terminology in 4D, refer to the **GET PRINTABLE MARGIN** command description.

To know the total size of the page, you can:

- either add the margins supplied by the **GET PRINTABLE MARGIN** command to the values returned by this command.
- or use the following syntax:

```
SET PRINTABLE MARGIN(0;0;0;0) ` Set the paper margin
GET PRINTABLE AREA(hPaper;wPaper) ` Paper size
```

# GET PRINTABLE MARGIN

GET PRINTABLE MARGIN ( left ; top ; right ; bottom )

| Parameter | Type | | Description |
|---|---|---|---|
| left | Longint | ⇦ | Left margin |
| top | Longint | ⇦ | Top margin |
| right | Longint | ⇦ | Right margin |
| bottom | Longint | ⇦ | Bottom margin |

## Description

The **GET PRINTABLE MARGIN** command returns the current values of the different margins defined using the **Print form**, **PRINT SELECTION** and **PRINT RECORD** commands.

The values are returned in pixels with respect to the paper edges.

It is possible to obtain the paper size as well as to calculate the printable area using the **GET PRINTABLE AREA** function.

### About Printable Margin Management

By default, the printing calculation in 4D is based on “printable margins”. The advantage of this system is that the forms adapt themselves automatically to the new printers (since they are positioned in the printable area). On the other hand, in the case of pre-printed forms, it was not possible to position the elements to be printed precisely because changing the printer can modify the printable margins.

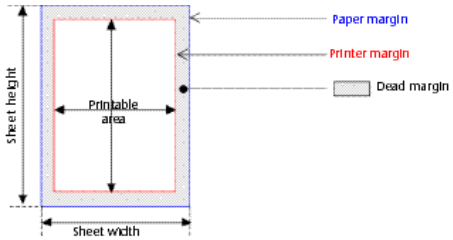
It is possible to base the form printing carried out using the **Print form**, **PRINT SELECTION** and **PRINT RECORD** commands on a fixed margin which is identical on each printer: the paper margins, i.e. the physical limits of the sheet. To do this, simply use the **GET PRINTABLE MARGIN**, **SET PRINTABLE MARGIN** and **GET PRINTABLE AREA** commands.

## About Printing Terminology

**Paper margin**: the paper margin corresponds to the physical limits of the sheet.

**Printer margin**: the printer margin is the margin beyond which the printer is incapable of printing (for material reasons: print rollers, printer head end-of-travel...). It varies from one printer to another and from one format to another.

**Dead margin**:this refers to the area located between the paper margin and the printer margin.



# Get printed height

Get printed height -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint |  | Position of the marker |

## Description

The **Get printed height** command returns the overall height (in pixels) of the section printed using the **Print form** command.

The value returned will be included between 0 (the top edge of the page) and the overall height returned by the **GET PRINTABLE AREA** command (the maximum size of the printable area).

If you print a new section using the **Print form** command, the height of the new section is added to this value. If the printable area available is insufficient to contain this section, a new page is generated and the value returned is 0.

The right and left printable margins, unlike the top and bottom margins (which may be defined using the **SET PRINTABLE MARGIN** command), do not influence the value returned.

**Note:** For more information regarding Printing management and terminology in 4D, refer to the **GET PRINTABLE MARGIN** command description.

# Is in print preview

Is in print preview -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | True = Print preview,<br>False = No print preview |

## Description

The **Is in print preview** command returns True if the **Preview on Screen** option is checked in the printing dialog box and False otherwise. This setting is local to the process.

Unlike the **Get print preview** command, **Is in print preview** returns the final value of the option, after the dialog box is validated by the user. So this command lets you determine with certainty whether printing actually takes place in "preview" mode.

## Example

This example takes all types of printing into account:

```
SET PRINT PREVIEW(True) //Print preview by default
PRINT SETTINGS
If(OK=1)
 //The user may have changed the print destination
   If(Is in print preview) // True if preview
      FORM SET OUTPUT([Invoices];"toScreen")
   Else
      FORM SET OUTPUT([Invoices];"toPrinter"
   End if
   OPEN PRINTING JOB
   ALL RECORDS([Invoices])
   PRINT SELECTION([Invoices];>)
   CLOSE PRINTING JOB
End if
```

# Level

Level -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Longint | Current break or header level |

## Description

**Level** is used to determine the current header or break level. It returns the level number during the On Header and On Printing Break events.

Level 0 is the last level to be printed and is appropriate for printing a grand total. **Level** returns 1 when 4D prints a break on the first sorted field, 2 when 4D prints a break on the second sorted field, and so on.

## Example

This example is a template for a form method. It shows each of the possible events that can occur while a summary report uses a form as an output form. **Level** is called when a header or a break is printed:

```
  ` Method of a form being used as output form for a summary report
$vpFormTable:=Current form table
Case of
  ` ...
  :(Form event=On Header)
  ` A header area is about to be printed
     Case of
        :(Before selection($vpFormTable->))
  ` Code for the first break header goes here
        :(Level=1)
  ` Code for a break header level 1 goes here
        :(Level=2)
  ` Code for a break header level 2 goes here
  ` ...
     End case
  :(Form event=On Printing Detail)
  ` A record is about to be printed
  ` Code for each record goes here
  :(Form event=On Printing Break)
  ` A break area is about to be printed
     Case of
        :(Level=0)
  ` Code for a break level 0 goes here
        :(Level=1)
  ` Code for a break level 1 goes here
  ` ...
     End case
  :(Form event=On Printing Footer)
     If(End selection($vpFormTable->))
  ` Code for the last footer goes here
     Else
  ` Code for a footer goes here
     End if
End case
```

# OPEN PRINTING JOB

OPEN PRINTING JOB
Does not require any parameters

## Description

The **OPEN PRINTING JOB** command opens a print job and stacks all the subsequent printing orders there until the **CLOSE PRINTING JOB** command is called. This command lets you control the print jobs and, more particularly, ensure that no other unexpected print job can be inserted into a printing sequence.

The **OPEN PRINTING JOB** command can be used with all the 4D printing commands, the quick report commands, and the printing commands of the 4D Write and 4D View plug-ins. On the other hand, this command is not compatible with the 4D Chart plug-in, as well as most third-party plug-ins.

When a print job is opened with this command, the printer is placed in “busy” mode until the printing is actually launched. If a noncompatible plug-in launches a print job in this context, the “printer busy” error is returned.

You must call the **CLOSE PRINTING JOB** command to terminate the print job and send the print document to the printer. If you omit this command, the print document will remain in the stack and the printer will not be available until you quit the 4D application.

The print job is local to the process but printing is performed at the global level. Only one print job can be open at a time in 4D, all processes taken together.

**OPEN PRINTING JOB** uses the current print settings (default settings or set using the **PAGE SETUP** and/or **SET PRINT OPTION** commands). The commands that modify the print settings must be called before **OPEN PRINTING JOB**. Otherwise, an error is generated.

# PAGE BREAK

PAGE BREAK {( * | > )}

| Parameter | Type | | Description |
|---|---|---|---|
| * \| > | | ⇨ | * Cancel printing job started with Print form, or > Force one printing job |

## Description

**PAGE BREAK** triggers the printing of the data that has been sent to the printer and ejects the page. **PAGE BREAK** is used with **Print form** (in the context of the On Printing Detail form event) to force page breaks and to print the last page created in memory. Do not use **PAGE BREAK** with the **PRINT SELECTION** command. Instead, use **Subtotal** or **BREAK LEVEL** with the optional parameter to generate page breaks.

The * and > parameters are both optional.

The * parameter allows you to cancel a print job started with the **Print form** command. Executing this command immediately stops the print job in progress.

**Note:** Under Windows, this mechanism can be disrupted by the spooling properties of the print server. If the printer is configured to start printing immediately, cancelling will not be effective. For the **PAGE BREAK**(*) command to operate correctly, it is preferable to choose the "Start printing after last page is spooled" property for the printer.

The > parameter modifies the way in which the **PAGE BREAK** command behaves. This syntax has two effects:

- It holds the print job open until the **PAGE BREAK** command is executed again without a parameter.
- It gives priority to the print job. No other printing can take place until the print job is finished.
  The second option is particularly useful when used with a spooled print job. The > parameter guarantees that the print job will be spooled to one file. This will reduce printing time.

**Note:** When screen printing, if the user clicks on Cancel in the print preview dialog box, the **PAGE BREAK** command sets the system variable OK to 0.

## Example 1

See example for the **Print form** command.

## Example 2

Refer to the example of the **SET PRINT MARKER** command.

# PAGE SETUP

PAGE SETUP ( {aTable ;} form )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table owning form, or Default table, if omitted |
| form | String | ⇒ | Form to use for page setup |

## Description

**PAGE SETUP** sets the page setup for the printer to that stored with *form*. The page setup is stored with the form when the form is saved in the Design environment.

In the following three cases, the printing dialog boxes are not displayed and the printing is performed with the default print settings:

- Calling **PRINT SELECTION** to which you pass the optional * parameter
- Calling **PRINT RECORD** to which you pass the optional * parameter
- Issuing a series of calls to **Print form** not preceeded by a call to **PRINT SETTINGS**.

Calling **PAGE SETUP** enables you, in this case, to skip the printing dialog boxes AND to use print settings other than the default ones.

## Example

Several (empty) forms are created for a table named [Design Stuff]. The form “PS100” is assigned a page setup with a scaling of 100%, the form “PS90” is assigned a page setup with a scaling of 90%, and so on. The following project method enables you to print the selection of a table using various scalings without having to specify the scaling in the printing dialog boxes (which are not displayed), each time:

```
  ` AUTOMATIC SCALED PRINTING project method
  ` AUTOMATIC SCALED PRINTING ( Pointer ; String {; Long } )
  ` AUTOMATIC SCALED PRINTING ( ->[Table]; "Output form" {; Scaling } )
If(Count parameters>=3)
   PAGE SETUP([Design Stuff];"PS"+String($3))
   If(Count parameters>=2)
      OUTPUT FORM($1->;$2)
   End if
End if
If(Count parameters>=1)
   PRINT SELECTION($1->;*)
Else
   PRINT SELECTION(*)
End if
```

Once this project method is written, you call it in this way:

```
  ` Look for current invoices
QUERY([Invoices];[Invoices]Paid=False)
  ` Print Summary Report in 90% reduction
AUTOMATIC SCALED PRINTING(->[Invoices];"Summary Report";90)
  ` Print Detailed Report in 50% reduction
AUTOMATIC SCALED PRINTING(->[Invoices];"Detailed Report";50)
```

# Print form

Print form ( {aTable ;} form {; area1 {; area2}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table owning the form, or Default table, if omitted |
| form | String | ⇒ | Form to print |
| area1 | Longint | ⇒ | Print marker, or Beginning area (if area2 is specified) |
| area2 | Longint | ⇒ | Ending area (if area1 specified) |
| Function result | Longint | ⇐ | Height of printed section |

## Description

**Print form** simply prints *form* with the current values of fields and variables of *aTable*. It is usually used to print very complex reports that require complete control over the printing process. **Print form** does not do any record processing, break processing or page breaks. These operations are your responsibility. **Print form** prints fields and variables in a fixed size frame only.

Since **Print form** does not issue a page break after printing the form, it is easy to combine different forms on the same page. Thus, **Print form** is perfect for complex printing tasks that involve different tables and different forms. To force a page break between forms, use the **PAGE BREAK** command. In order to carry printing over to the next page for a form whose height is greater than the available space, call the **CANCEL** command before the **PAGE BREAK** command.

Three different syntaxes may be used:

- **Detail area printing**

Syntax:

```
height:=Print form(myTable;myForm)
```

In this case, **Print form** only prints the Detail area (the area between the Header line and the Detail line) of the form.

- **Form area printing**

Syntax:

```
height:=Print form(myTable;myForm;marker)
```

In this case, the command will print the section designated by the *marker*. Pass one of the constants of the **Form Area** theme in the marker parameter:

| Constant | Type | Value |
|---|---|---|
| Form break0 | Longint | 300 |
| Form break1 | Longint | 301 |
| Form break2 | Longint | 302 |
| Form break3 | Longint | 303 |
| Form break4 | Longint | 304 |
| Form break5 | Longint | 305 |
| Form break6 | Longint | 306 |
| Form break7 | Longint | 307 |
| Form break8 | Longint | 308 |
| Form break9 | Longint | 309 |
| Form detail | Longint | 0 |
| Form footer | Longint | 100 |
| Form header | Longint | 200 |
| Form header1 | Longint | 201 |
| Form header10 | Longint | 210 |
| Form header2 | Longint | 202 |
| Form header3 | Longint | 203 |
| Form header4 | Longint | 204 |
| Form header5 | Longint | 205 |
| Form header6 | Longint | 206 |
| Form header7 | Longint | 207 |
| Form header8 | Longint | 208 |
| Form header9 | Longint | 209 |

- **Section printing**

Syntax:

```
height:=Print form(myTable;myForm;areaStart;areaEnd)
```

In this case, the command will print the section included between the *areaStart* and *areaEnd* parameters. The values entered must be expressed in pixels.

The value returned by **Print form** indicates the height of the printable area. This value will be automatically taken into account by the **Get printed height** command.

The printer dialog boxes do not appear when you use **Print form**. The report does not use the print settings that were assigned to the form in the Design environment. There are two ways to specify the print settings before issuing a series of calls to **Print form**:

- Call **PRINT SETTINGS**. In this case, you let the user choose the settings.
- Call **PAGE SETUP**. In this case, print settings are specified programmatically.

**Print form** builds each printed page in memory. Each page is printed when the page in memory is full or when you call **PAGE BREAK**. To ensure the printing of the last page after any use of **Print form**, you must conclude with the **PAGE BREAK** command (except in the context of an **OPEN PRINTING JOB**, see note). Otherwise, if the last page is not full, it stays in memory and is not printed.

**Warning:** If the command is called in the context of a printing job opened with **OPEN PRINTING JOB**, you must NOT call **PAGE BREAK** for the last page because it is automatically printed by the **CLOSE PRINTING JOB** command. If you call **PAGE BREAK** in this case, a blank page is printed.

This command prints external areas and objects (for example, 4D Write or 4D View areas). The area is reset for each execution of the command.

**Warning:** Subforms are not printed with **Print form**. To print only one form with such objects, use **PRINT RECORD** instead.

**Print form** generates only one On Printing Detail event for the form method.

**4D Server:** This command can be executed on 4D Server within the framework of a stored procedure. In this context:

- Make sure that no dialog box appears on the server machine (except for a specific requirement).

- In the case of a problem concerning the printer (out of paper, printer disconnected, etc.), no error message is generated.

## Example 1

The following example performs as a **PRINT SELECTION** command would. However, the report uses one of two different forms, depending on whether the record is for a check or a deposit:

```
QUERY([Register]) ` Select the records
If(OK=1)
   ORDER BY([Register]) ` Sort the records
   If(OK=1)
      PRINT SETTINGS ` Display Printing dialog boxes
      If(OK=1)
         For($vlRecord;1;Records in selection([Register]))
            If([Register]Type ="Check")
               Print form([Register];"Check Out") ` Use one form for checks
            Else
               Print form([Register];"Deposit Out") ` Use another form for deposits
            End if
            NEXT RECORD([Register])
         End for
         PAGE BREAK ` Make sure the last page is printed
      End if
   End if
End if
```

## Example 2

Refer to the example of the **SET PRINT MARKER** command.

# PRINT LABEL

PRINT LABEL ( {aTable }{;}{ document {; * | >}} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to print, or Default table, if omitted |
| document | String | ⇒ | Name of disk label document |
| * \| > | | ⇒ | * to suppress the printing dialog boxes, or > to not reinitialize print settings |

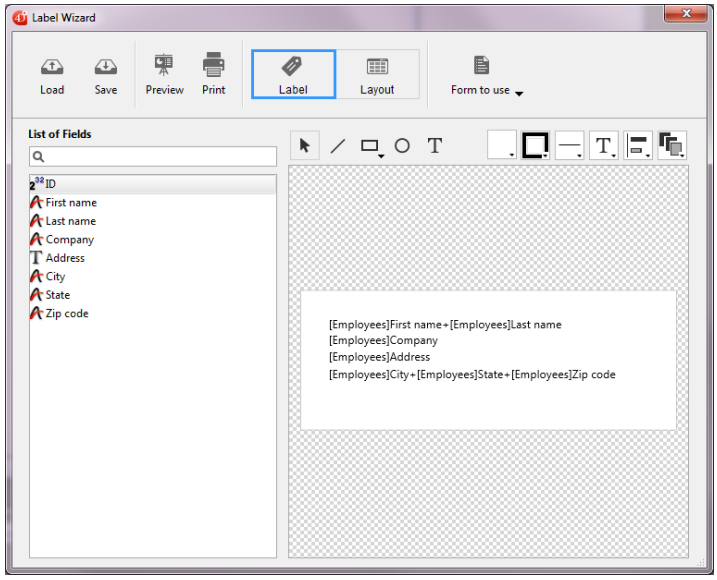
## Description

**PRINT LABEL** enables you to print labels with the data from the selection of *aTable*.

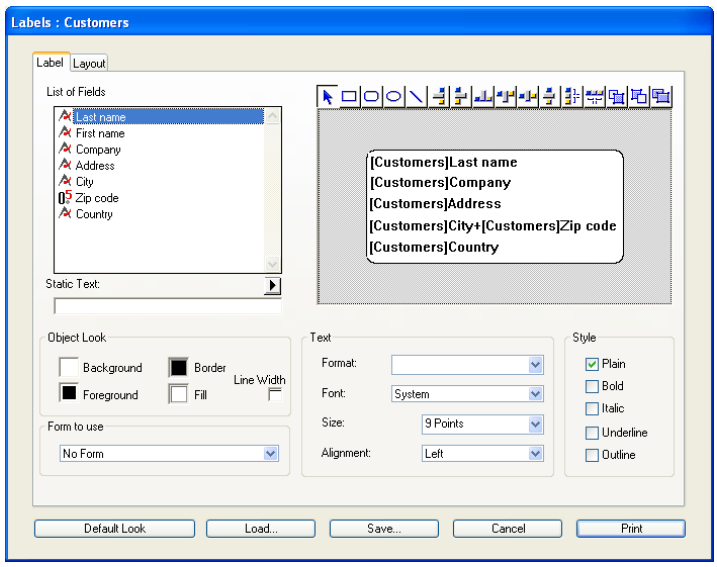
If do not specify the *document* parameter, **PRINT LABEL** prints the current selection of *aTable* as labels, using the current output form. You cannot use this command to print subforms. For details about creating forms for labels, refer to the 4D Design Reference manual.

If you specify the *document* parameter, **PRINT LABEL** enables you to access the Label Wizard (shown below) or to print an existing Label document stored on disk. See the following discussion.

64-bit version:




32-bit version:




**Compatibility note:** The 32-bit version of the label editor only supports ASCII characters. If you need to use a more extended character set, you have to print labels using the current output form.

By default, **PRINT LABEL** displays the printer dialog boxes (in 4D 32-bit versions) or the Print job dialog box (in 4D 64-bit versions) before printing. If the user cancels either of the printer dialog boxes, the command is canceled and the labels are

not printed.

You can suppress these dialog boxes by using either the optional asterisk (*) parameter or the optional “greater than” (>) parameter:

- The * parameter causes a print job using the current print parameters.
- Furthermore, the > parameter causes a print job without reinitializing the current print parameters. This setting is useful for executing several successive calls to **PRINT LABEL** (ex. inside a loop) while maintaining previously set customized print parameters. For an example of use of this parameter, refer to the **PRINT RECORD** command description.

Note that this parameter has no effect if the Label Wizard is involved.

If the Label Wizard is not involved, the OK variable is set to 1 if all labels are printed; otherwise, it is set to 0 (zero) (i.e., if user clicked **Cancel** in the printing dialog boxes).

If you specify the *document* parameter, the labels are printed with the label setup defined in *document*. If *document* is an empty string (""), **PRINT LABEL** will present an Open File dialog box so the user can specify the file to use for the label setup. If *document* is the name of a document that does not exist (for example, pass *char(1)* in *document)*, the Label Wizard is displayed and the user can define the label setup.

**Note:** If the *table* has been declared “invisible” in Design mode, the Label Wizard will not be displayed.

**4D Server:** This command can be executed on 4D Server within the framework of a stored procedure. In this context:

- Make sure that no dialog box appears on the server machine (except for a specific requirement). To do this, it is necessary to call the command with the * or > parameter.
- The syntax which makes the label editor appear does not work with 4D Server; in this case, the system variable OK is set to 0.
- In the case of a problem concerning the printer (out of paper, printer disconnected, etc.), no error message is generated.

## Example 1

The following example prints labels using the output form of a table. The example uses two methods. The first is a project method that sets the correct output form and then prints labels:

```
ALL RECORDS([Addresses]) ` Select all records
FORM SET OUTPUT([Addresses];"Label Out") ` Select the output form
PRINT LABEL([Addresses]) ` Print the labels
FORM SET OUTPUT([Addresses];"Output") ` Restore default output form
```

The second method is the form method for the form *"Label Out"*. The form contains one variable named *vLabel*, which is used to hold the concatenated fields. If the second address field (Addr2) is blank, it is removed by the method. Note that this task is performed automatically with the Label Wizard. The form method creates the label for each record:

```
  ` [Addresses]; "Label Out" form method
Case of
   :(Form event=On Load)
      vLabel:=[Addresses]Name1+" "+[Addresses]Name2+Char(13)+[Addresses]Addr1+Char(13)
      If([Addresses]Addr2 #"")
         vLabel:=vLabel+[Addresses]Addr2+Char(13)
      End if
      vLabel:=vLabel+[Addresses]City+", "+[Addresses]State+" "+[Addresses]ZipCode
End case
```

## Example 2

The following example lets the user query the [People] table, and then automatically prints the labels “My Labels”:

```
QUERY([People])
If(OK=1)
   PRINT LABEL([People];"My Labels";*)
End if
```

## Example 3

The following example lets the user query the [People] table, and then lets the user choose the labels to be printed:

```
QUERY([People])
If(OK=1)
   PRINT LABEL([People];"")
End if
```

## Example 4

The following example lets the user query the [People] table, and then displays the Label Wizard so the user can design, save, load and print any labels:

```
QUERY([People])
If(OK=1)
   PRINT LABEL([People];Char(1))
End if
```

# Print object

Print object ( {* ;} object {; posX {; posY {; width {; height}}}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| posX | Longint | ⇒ | Horizontal location of object |
| posY | Longint | ⇒ | Vertical location of object |
| width | Longint | ⇒ | Width of object (pixels) |
| height | Longint | ⇒ | Height of object (pixels) |
| Function result | Boolean | ⇐ | True = object entirely printed; otherwise False |

## Description

The **Print object** command lets you print the form object(s) designated by the *object* and * parameters, at the location set by the *posX* and *posY* parameters.

Before calling the **Print object** command, you must designate the table or project form containing the objects to be printed, using the **FORM LOAD** command.

If you pass the optional * parameter, you indicate that the object parameter is an object name (character string). If you do not pass the * parameter, you indicate that object is a variable. In this case, you pass a variable reference (object type only) instead of a string.

The *posX* and *posY* parameters specify the starting point for printing the object(s). These values must be expressed in pixels. If these parameters are omitted, the object will be printed according to its location in the form.

The *width* and *height* parameters are used to specify the width and height of the form object. The **Print object** command does not manage objects of variable size. You must use the **OBJECT GET BEST SIZE** command to manage the size of objects. You can also use the **OBJECT GET BEST SIZE** command to find out the most appropriate size for objects containing text. Similarly, **Print object** will not cause automatic page breaks. You must manage them according to your needs.

You can use 4D commands to modify object properties (color, size, etc.) on the fly.

The command returns True if the object has been completely printed and False if this is not the case; in other words, if it was not able to print all the data associated with the object within the set framework. Typically, the command returns False when printing a list box if all the rows of the list box could not be printed. In this case, you can simply call the **Print object** command repeatedly until it returns True: a specific mechanism automatically causes the contents of the object to scroll after each call.

**Notes:**

- In the current version of 4D, only list box type objects have this mechanism (the command always returns True for any other type of object). In forthcoming versions of 4D, this functioning will be extended to other objects with variable contents.
- The **LISTBOX GET PRINT INFORMATION** command lets you check the status of the printing during the operation.

The **Print object** command can only be used in the context of a print job opened beforehand with the **OPEN PRINTING JOB** command. If it is not called in this context, the command does nothing. Several **Print object** commands can be called in the same print job.

**Note:** Hierarchical lists, subforms and Web areas cannot be printed.

## Example 1

Example for printing ten objects in a form:

```
PRINT SETTINGS
If(OK=1)
   OPEN PRINTING JOB
   If(OK=1)
      FORM LOAD("PrintForm")
      x:=100
      y:=50
      GET PRINTABLE AREA(hpaper;wpaper)
```

```
      For($i;1;10)
         OBJECT GET BEST SIZE(*;"Obj"+String($i);bestwidth;bestheight)
         $end:=Print object(*;"Obj"+String($i))
         y:=y+bestheight+15
         If(y>hpaper)
            PAGE BREAK(>)
            y:=50
         End if
      End for
   End if
   CLOSE PRINTING JOB
End if
```

## Example 2

Example of printing a complete list box:

```
Repeat
   $end:=Print object(*;"mylistbox")
Until($end)
```

# PRINT OPTION VALUES

PRINT OPTION VALUES ( option ; namesArray {; info1Array {; info2Array}} )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ⇒ | Option number |
| namesArray | Text array | ⇐ | Names of values |
| info1Array | Longint array | ⇐ | Values (1) of the option |
| info2Array | Longint array | ⇐ | Values (2) of the option |

## Description

In *namesArray*, the **PRINT OPTION VALUES** command returns a list of value names available for the print *option* defined. Optionally, you can retrieve information for each value in *info1Array* and *info2Array*.

The *option* parameter allows you to specify the option to get. You must pass one of the following constants of the "**Print Options**" theme (options able to return lists of value names):

| Constant | Type | Value | Comment |
|---|---|---|---|
| Paper option | Longint | 1 | If you use only *value1*, it contains the name of the paper. If you use both parameters, *value1* contains the paper width and *value2* contains the paper height. The width and height are expressed in screen pixels. Use the **PRINT OPTION VALUES** command to get the name, height and width of all the paper formats offered by the printer. |
| Paper source option | Longint | 5 | (Windows only) *value1* only: number corresponding to the index, in the array of trays returned by the **PRINT OPTION VALUES** command, of the paper tray to be used. This option can only be used under Windows. |

After command execution, the *namesArray* array as well as, where applicable, the *info1Array* and *info2Array* arrays will be filled in by the command with the names and information of the available values.

If you pass value 1 (Paper option) in the *option* parameter, the command will return the following information:

- in *namesArray*, the names of the available paper formats;
- in *info1Array*, the heights of each paper format;
- in *info2Array*, the widths of each paper format.

**Note:** In order to obtain this information, the print driver must have access to a valid PPD (PostScript Printer Description) file for the printer.

In order to apply a specific paper format using the **SET PRINT OPTION** command, you can either pass one of the values of *namesArray*, the corresponding values of *info1Array* and *info2Array*.

If you pass value 5 (Paper source option) in the *option* parameter, the command returns the names of the different trays available in *namesArray*, and their internal Windows ID numbers in *info1Array* (*info2Array* remains empty).

The order of the values in the arrays is defined by the print driver. To indicate a tray using the **SET PRINT OPTION** command, you must pass the index, as found in the *namesArray* or *info1Array* arrays, of the element desired.

**Note:** This option can only be used under Windows.

For more information on the different print options, refer to the description of the **SET PRINT OPTION** and **GET PRINT OPTION** commands.

All the information returned by these commands is supplied by the operating system. Refer to the documentation of your system for more details about specific options.

**Note:** The **PRINT OPTION VALUES** command only operates with PostScript printers.

# PRINT RECORD

PRINT RECORD ( {aTable}{;}{* | >} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to print the current record or Default table if omitted |
| * \| > | Operator | ⇒ | * to suppress the printer dialog boxes, or > to not reinitialize print settings |

## Description

**PRINT RECORD** prints the current record of *aTable*, without modifying the current selection. The current output form is used for printing. If there is no current record for *aTable*, **PRINT RECORD** does nothing.

You can print subforms with the **PRINT RECORD** command. This is not possible with **Print form**.

**Note:** If there are modifications to the record that have not been saved, this command prints the modified field values, not the field values located on disk.

By default, **PRINT RECORD** displays the printer dialog boxes (in 4D 32-bit versions) or the Print job dialog box (in 4D 64-bit versions) before printing. If the user cancels either of the printer dialog boxes, the command is canceled and the record is not printed.

You can suppress these dialog boxes by using either the optional asterisk (*) parameter or the optional “greater than” (>) parameter:

- The * parameter causes a print job using the current print parameters (default parameters or those defined by the **PAGE SETUP** and/or **SET PRINT OPTION** commands).
- Furthermore, the > parameter causes a print job without reinitializing the current print parameters. This setting is useful for executing several successive calls to **PRINT RECORD** (e.g. inside a loop) while maintaining previously set customized print parameters.

**4D Server:** This command can be executed on 4D Server within the framework of a stored procedure. In this context:

- Make sure that no dialog box appears on the server machine (except for a specific requirement). To do this, it is necessary to call the command with the * or > parameter.
- In the case of a problem concerning the printer (out of paper, printer disconnected, etc.), no error message is generated.

**Warning:** Do not use the **PAGE BREAK** command with **PRINT RECORD**. **PAGE BREAK** is exclusively reserved for use in combination with the **Print form** command.

## Example 1

The following example prints the current record of the [Invoices] table. The code is contained in the object method of a **Print** button on the input form. When the user clicks the button, the record is printed using an output form designed for this purpose.

```
FORM SET OUTPUT([Invoices];"Print One From Data Entry") ` Select the right output form for printing
PRINT RECORD([Invoices];*) ` Print Invoices as it is (without showing the printing dialog boxes)
FORM SET OUTPUT([Invoices];"Standard Output") ` Restore the previous output form
```

## Example 2

The following example prints the same current record in two different forms. The code is contained in the object method of a **Print** button on the input form. You want to set customized print parameters and then use them in the two forms.

```
PRINT SETTINGS `User defines print parameters
If(OK=1)
   FORM SET OUTPUT([Employees];"Detailed") `Use the first print form
   PRINT RECORD([Employees];>) `Print using user-defined parameters
   FORM SET OUTPUT([Employees];"Simple") `Use the second print form
   PRINT RECORD([Employees];>) `Print using user-defined parameters
```

```
      FORM SET OUTPUT([Employees];"Output") `Restore default output form
   End if
```

# PRINT SELECTION

PRINT SELECTION ( {aTable}{;}{* | >} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to print the selection, or Default table, if omitted |
| * \| > | Operator | ⇒ | * to delete the printing dialog boxes, or > to not reinitialize print settings |

## Description

**PRINT SELECTION** prints the current selection of *aTable*. The records are printed with the current output form of the table in the current process. **PRINT SELECTION** performs the same action as the Print menu command in the Design environment. If the selection is empty, **PRINT SELECTION** does nothing.

By default, **PRINT SELECTION** displays the printer dialog boxes (in 4D 32-bit versions) or the Print job dialog box (in 4D 64-bit versions) before printing. If the user cancels either of the printer dialog boxes, the command is canceled and the report is not printed.

You can delete these dialog boxes by using either the optional asterisk (*) parameter or the optional “greater than” (>) parameter:

- The * parameter causes a print job using the current print parameters (default parameters or those defined by the **PAGE SETUP** and/or **SET PRINT OPTION** commands).
- Furthermore, the > parameter causes a print job without reinitializing the current print parameters. This setting is useful for executing several successive calls to **PRINT SELECTION** (e.g., inside a loop) while maintaining previously set customized print parameters. For an example of the use of this parameter, refer to the **PRINT RECORD** command description.

During printing, the output form method and/or the form’s object methods are executed depending on the events that are enabled for the form and objects using the Property List window in the Design environment, as well as on the events actually occurring:

- An On Header event is generated just before a header area is printed.
- An On Printing Detail event is generated just before a record is printed.
- An On Printing Break event is generated just before a break area is printed.
- An On Printing Footer event is generated just before a footer is printed.

You can check whether **PRINT SELECTION** is printing the first header by testing **Before selection** during an On Header event. You can also check for the last footer, by testing **End selection** during an On Printing Footer event. For more information, see the description of these commands, as well as those of **Form event** and **Level**.

To print a sorted selection with subtotals or breaks using **PRINT SELECTION**, you must first sort the selection. Then, in each Break area of the report, include a variable with an object method that assigns the subtotal to the variable. You can also use statistical and arithmetical functions like **Sum** and **Average** to assign values to variables. For more information, see the descriptions of **Subtotal**, **BREAK LEVEL** and **ACCUMULATE**.

**Warning:** Do not use the **PAGE BREAK** command with the **PRINT SELECTION** command. **PAGE BREAK** is to be used with the **Print form** command.

After a call to **PRINT SELECTION**, the OK variable is set to 1 if the printing has been completed. If the printing was interrupted, the OK variable is set to 0 (zero) (i.e., the user clicked Cancel in the printing dialog boxes).

**4D Server:** This command can be executed on 4D Server within the framework of a stored procedure. In this context:

- Make sure that no dialog box appears on the server machine (except for a specific requirement). To do this, it is necessary to call the command with the * or > parameter.
- In the case of a problem concerning the printer (out of paper, printer disconnected, etc.), no error message is generated.

## Example

The following example selects all the records in the [People] table. It then uses the **DISPLAY SELECTION** command to display the records and allows the user to highlight the records to print. Finally, it uses the selected records with the **USE SET** command, and prints them with **PRINT SELECTION**:

```
ALL RECORDS([People]) ` Select all records
DISPLAY SELECTION([People];*) ` Display the records
USE SET("UserSet") ` Use only records picked by user
PRINT SELECTION([People]) ` Print the records that the user picked
```

# PRINT SETTINGS

PRINT SETTINGS {( dialType )}

| Parameter | Type | | Description |
|---|---|---|---|
| dialType | Longint | ⇨ | Dialog box(es) to be displayed |

## Description

**PRINT SETTINGS** displays either one or two print settings dialog boxes. This command must be called before a series of **Print form** commands or the **OPEN PRINTING JOB** command.

The optional *dialType* parameter can be used to configure the display of the printing dialog boxes. You can use one of the following constants from the **Print Options** theme. The displayed dialog box(es) will actually depend on your 4D version, as shown in the following table:

| dialType value (constant) | 4D 64-bit | 4D 32-bit |
|---|---|---|
| 0 or omitted | Print job | Page setup+Print job |
| 1 (Page setup dialog) | Page setup | Page setup |
| 2 (Print dialog) | Print job (= 0 or omitted) | Print job |

**Note:** The Print Job dialog box contains a **Preview on Screen** check box that allows the user to preview their print job. You can preset or reset this option by calling **SET PRINT PREVIEW** before calling **PRINT SETTINGS**.

## Example

See example for the command **Print form**.

## System variables and sets

If the user clicks OK in both dialog boxes, the OK system variable is set to 1. Otherwise, the OK system variable is set to 0.

# Print settings to BLOB

Print settings to BLOB ( printSettings ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| printSettings | BLOB | ⇦ | Current print settings |
| Function result | Longint | ⇨ | Status code: 1=Operation successful, 0=No current printer |

## Description

The **Print settings to BLOB** command saves the current 4D print settings in the *printSettings* BLOB. The *printSettings* parameter stores all the settings used for printing:

- Layout parameters such as paper, orientation, scale, etc.
- Print parameters such as number of copies, paper source, etc.

This command must be used in conjunction with the **BLOB to print settings** command. These commands allow you to save a user's current print settings and reload them subsequently so that users will not need to specify their parameters each time they start a print job. In addition, it allows you to save "private" printer settings (specific to the printer driver) that are not available as standard printing parameters.

The BLOB generated must not be modified by programming; it can only be used by the **BLOB to print settings** command.

The command returns 1 if the BLOB has been generated correctly, and 0 if no current printer is selected.

### Windows / OS X

The *printSettings* BLOB can be saved and read on both platforms. However, even if some print settings are common, some others are platform-specific and depend on the drivers and system versions. If the same *printSettings* BLOB is shared between both platforms, you may lost information parts.
When used in an heterogeneous environment, in order to restore the maximum settings available for each platform (and not only the common part), it is recommended that you handle two *printSettings* BLOBs, one for each platform.

## Example

You want to store the current print settings to disk:

```
C_BLOB(curSettings)
PRINT SETTINGS //displays print settings dialog to the user
If(OK=1)
   $err:=Print settings to BLOB(curSettings)
   If($err=1)
      BLOB TO DOCUMENT(Get 4D folder+"current4Dsettings.blob":curSettings)
   Else
      ALERT("No selected printer")
   End if
End if
```

# PRINTERS LIST

PRINTERS LIST ( namesArray {; altNamesArray {; modelsArray}} )

| Parameter | Type | | Description |
|---|---|---|---|
| namesArray | Text array | ⇦ | Printer names |
| altNamesArray | Text array | ⇦ | Windows: Printer locations Mac OS: Custom printer names |
| modelsArray | Text array | ⇦ | Printer models |

## Description

The **PRINTERS LIST** command fills in the array(s) passed as parameter(s) with the names as well as, optionally, the locations or custom names and models of the available printers for the machine.

**Note:** If the printers are managed using a print server (spooler), the complete access path (under Windows) or the name of the spooler (under Mac OS) is returned.

Pass the name of a text array in the *namesArray* parameter. After command execution, this array will contain the names of available printers. Under Mac OS, this will be the fixed “system” names.

You can pass a second optional array, *altNamesArray*. The contents of this array will depend on the platform:

- Under Windows, for each printer, you get its network location (or local port).
- Under Mac OS, for each printer, you get its custom name, which can be modified by the user. This name can be used, for example, in dialog boxes.

The optional *modelsArray* parameter is used to get the model of each printer. (**Note:** This parameter is not supported on 32-bit versions of 4D for Mac).

Use the **SET CURRENT PRINTER** and **Get current printer** commands to modify or get the selected printer in 4D. You must pass them the names returned in the first array (*namesArray*)

Under Windows, the name of a printer can be modified manually at the operating system level. On the other hand, its location and model type are linked to its physical characteristics. Therefore, you can use the optional array values to check the characteristics of the selected printer — typically, you can check that all the client machines use the same printer.

Under Mac OS, this check can be carried out using the name of the printer (name of the print server), which is the same for each machine that is connected.

## System variables and sets

The system variable OK is set to 1 if the command has been executed correctly; otherwise, it is set to 0 and the arrays are returned empty.

# Printing page

Printing page -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Page number of page currently being printed |

## Description

**Printing page** returns the printing page number. It can be used only when you are printing with **PRINT SELECTION** or the Print menu in the Design environment.

## Example

The following example changes the position of the page numbers on a report so that the report can be reproduced in a double-sided format. The form for the report has two variables that display page numbers. A variable in the lower-left corner (*vLeftPageNum*) will print the even page numbers. A variable in the lower-right corner (*vRightPageNum*) will print the odd page numbers. The example tests for even pages, then clears and sets the appropriate variables:

```
Case of
   :(Form event=On Printing Footer)
      If((Printing page% 2)=0) ` Modulo is 0, it is an even page
         vLeftPageNum:=String(Printing page) ` Set the left page number
         vRightPageNum:="" ` Clear the right page number
      Else ` Otherwise it is an odd page
         vLeftPageNum:="" ` Clear the left page number
         vRightPageNum:=String(Printing page) ` Set the right page number
      End if
End case
```

# SET CURRENT PRINTER

SET CURRENT PRINTER ( printerName )

| Parameter | Type | | Description |
|---|---|---|---|
| printerName | String | ⇒ | Name of printer to be used |

## Description

The **SET CURRENT PRINTER** command is designates the printer to be used for printing with the current 4D application.

Pass the name of the printer to be selected in the *printerName* parameter. To get a list of available printers, use the **PRINTERS LIST** command beforehand.
If you pass an empty string in *printerName*, the current printer defined in the system will be used.

**SET CURRENT PRINTER** allows you to designate the generic PDF printer of the system in order to print PDFs. The value to use depends on the version of the OS as well as that of 4D.

- **Windows 8 and previous versions:**
  4D relies on the PDFCreator driver to facilitate the printing of PDF documents under Windows (see the **Integration of PDFCreator driver under Windows** section). To print a PDF document, in the *printerName* parameter, pass the name of the virtual printer that was installed by the PDFCreator driver. By default, the name of the virtual printer is "PDFCreator". However, this name may have been modified when the driver was installed. In order for 4D to automatically look for and use the name of the virtual printer, even if it has been customized, in the *printerName* parameter you must pass the following constant (**Print Options** theme):

| Constant | Type | Value |
|---|---|---|
| PDFCreator Printer name | String | PDFCreator |

- **Starting with Windows 10:**
  Windows 10 includes a native PDF printer driver, which allows 4D to create PDFs directly without it being necessary to use a third-party driver such as PDFCreator.
  The name of the driver is "Microsoft Print to PDF" (see the example found in the **Integration of PDFCreator driver under Windows** section).

- **Under OS X and starting with Windows 10 (4D Developer Edition v15 R5 64-bit and higher):**
  A constant found in the **Print Options** theme allows you to designate the generic PDF printer automatically, regardless of the platform. This facilitates the writing of generic code.

| Constant | Type | Value | Comment |
|---|---|---|---|
| Generic PDF driver | String | _4d_pdf_printer | **Note:** This function is not available in 32-bit versions of 4D.<br>○ On OS X, sets the current printer to the default driver. This driver is invisible; it is not found in the list returned by **PRINTERS LIST**. Note that a PDF document path must have been set using **SET PRINT OPTION**, otherwise error 3107 is returned.<br>○ On Windows, sets the current printer to the Windows PDF driver (named "Microsoft Print to PDF"). This constant is only available in Windows 10 with the PDF option installed. In other Windows versions, or if no PDF driver is available, the command does nothing and the *OK* variable is set to 0. |

The **SET CURRENT PRINTER** command must be called before **SET PRINT OPTION**, so that the options available correspond to the selected printer. On the other hand, **SET CURRENT PRINTER** must be called after **PAGE SETUP**, otherwise the print settings are lost.

This command can be used with the **PRINT SELECTION**, **PRINT RECORD**, **Print form**, and **QR REPORT** commands, and is applied to all 4D printing, including that in Design mode.
It is imperative for print commands to be called with the > parameter (where applicable) so that the specified settings are not lost.

## System variables and sets

If printer selection is carried out correctly, the system variable OK is set to 1. Should the opposite occur (for instance if the designated printer is not found), the system variable OK is set to 0 and the current printer remains unchanged.

## Example

Creation of a PDF document under Windows 10 with 4D Developer Edition 64-bit:

```
C_TEXT($pdfpath)
$pdfpath:=System folder(Desktop)+"test.pdf"
SET CURRENT PRINTER(Generic PDF driver)
SET PRINT OPTION(Destination option;2;$pdfpath)
ALL RECORDS([Table_1])
PRINT SELECTION([Table_1];*)
SET CURRENT PRINTER("")
```

# SET PRINT MARKER

SET PRINT MARKER ( markNum ; position {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| markNum | Longint | ⇒ | Marker number |
| position | Longint | ⇒ | New position for the marker |
| * | Operator | ⇒ | If passed = move subsequent markers If omitted = do not move subsequent markers |

## Description

The **SET PRINT MARKER** command enables the definition of the marker position during printing. Combined with the **Get print marker**, **OBJECT MOVE** or **Print form** commands, this command allows you to adjust the size of the print areas.

**SET PRINT MARKER** can be used in two contexts:

- during the On Header form event, in the context of **PRINT SELECTION** and **PRINT RECORD** commands.
- during the On Printing Detail form event, in the context of the **Print form** command. This operation facilitates the printing of customized reports (see example).

The effect of the command is limited to printing; no modification appears on the screen. The modifications made to the forms are not saved.

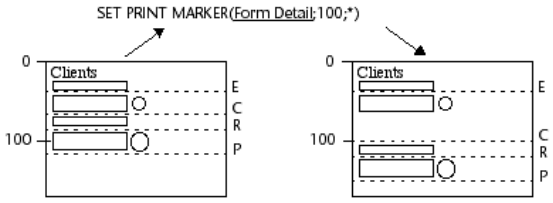
Pass one of the constants of the **Form Area** theme in the *markNum* parameter:

| Constant | Type | Value |
|---|---|---|
| Form break0 | Longint | 300 |
| Form break1 | Longint | 301 |
| Form break2 | Longint | 302 |
| Form break3 | Longint | 303 |
| Form break4 | Longint | 304 |
| Form break5 | Longint | 305 |
| Form break6 | Longint | 306 |
| Form break7 | Longint | 307 |
| Form break8 | Longint | 308 |
| Form break9 | Longint | 309 |
| Form detail | Longint | 0 |
| Form footer | Longint | 100 |
| Form header | Longint | 200 |
| Form header1 | Longint | 201 |
| Form header10 | Longint | 210 |
| Form header2 | Longint | 202 |
| Form header3 | Longint | 203 |
| Form header4 | Longint | 204 |
| Form header5 | Longint | 205 |
| Form header6 | Longint | 206 |
| Form header7 | Longint | 207 |
| Form header8 | Longint | 208 |
| Form header9 | Longint | 209 |

In *position*, pass the new position desired, expressed in pixels.

If you pass the optional * parameter, all the markers located below the marker specified in *markNum* will be moved the same number of pixels and in the same direction as this marker when the command is executed. **Warning:** in this case, any objects present in the areas located below the marker are also moved.

When the * parameter is used, it is possible to position the *markNum* marker beyond the initial position of the markers that follow it — these latter markers will be moved simultaneously.

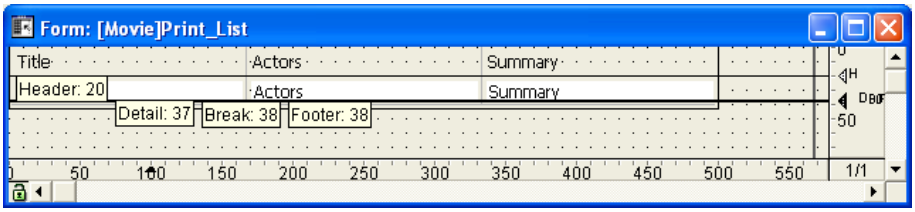



**Notes:**

- This command modifies only the existing marker position. It does not allow the addition of markers. If you designate a marker that does not exist in the form, the command will not do anything.
- The print marker mechanism in the Design mode is retained: a marker cannot go any higher than the one that precedes it, nor any lower than the one that follows it (when the * parameter is not used).

## Example

This complete example enables you to generate the printing of a three-column report, the height of each row being calculated on the fly according to the contents of the fields.
The output form used for printing is as follows:




The On Printing Detail form event was selected for the form (keep in mind that no matter what area is printed, the **Print form** command only generates this type of form event).
For each record, the row height must be adapted according to the contents of the "Actors" or "Summary" column (column having the most content). Here is the desired result:

| Title | Actors | Summary |
|---|---|---|
| The Avengers | Ralph Fiennes, Uma Thurman, Sean Connery, Jim Broadbent, Patrick Macnee, Fiona Shaw, Eddie Izzard, Eileen Atkins | "The Avengers", the popular TV series from the sixties, is brought to the big screen with a mix of humour, retro fashion and action. Ralph Fiennes plays the role of well dressed John Steed and Uma Thurman is the beautiful Emma Peel dressed in a jumpsuit. Our two special agents fight crime in style. Sean Connery plays Sir August De Wynter, an evil genious that wants to take over the world with his high-tech weather machine. Unleashing snow storms or heatwaves, and armed with remote-controlled killer bees, this nut is a real menance to society. But all these climate changes won't stop our two heroes. A cup of tea, anyone? |
| 20,000 Leagues Under The Sea | | "The year is 1868, when New England's fishing harbors are the scene for a creature of unknown origin destroying ships at sea. It is the job of Professor Pierre Aronnax, a marine expert, and Ned Land, the iron-willed sailor, to learn the truth of the monster roaming the seas. The great novelist, Jules Verne, described this perilous journey to the darkest depths of the sea with Captain Nemo aboard the Nautilus." |
| The Adventures Of Ichabod And Mr. Toad (Walt Disney C | Bing Crosby, Basil Rathbone, Eric Blore, Pat O'Malley, John McLeish, Colin Campbell, Campbell Grant, Claud Allister | Hang on for the wild motorcar ride of J. Thaddeus Toad as he drives his friends Mole, Rat and Angus MacBadger into a worried frenzy! Then meet the spindly Ichabod Crane, who dreams of sweeping beautiful Katrina Van Tassel off her feet, despite opposition from town bully Brom Bones, who also has his eye on Katrina. The comic rivalry introduces Ichabod to the legend of the Headless Horseman resulting in a heart-thumping climax. Wonderfully narrated by Basil Rathbone and Bing Crosby, The Adventures Of Ichabod And Mr. Toad brims with high-spirited adventure, brilliant animation and captivating music you'll want to share with your family time and again. |
| Mission To Mars | Gary Sinise, Tim Robbins, Don Cheadle, Jerry O'Connell, Connie Nielsen | From the director of Mission Impossible comes the thrilling, eye-popping science-fiction adventure Mission To Mars - starring Gary Sinise (Snake Eyes) and Tim Robbins (Austin Powers: The Spy Who Shagged Me). The year is 2020, and the first manned mission to Mars, commanded by Luke Graham (Don Cheadle - Out Of Sight), lands safely on the red planet. But the Martian landscape harbors a bizarre and shocking secret that leads to a mysterious disaster so catastrophic, it decimates the crew. Haunted by a cryptic last message from Graham, NASA launches the Mars Recovery Mission to investigate and bring back survivors - if there are any. Confronted with nearly insurmountable dangers, but propelled by deep friendship, the team finally lands on Mars and makes a discovery so amazing, it takes your breath away. Mission To Mars is an action-packed rocket ride that will enthrall you with its stunning special effects and keep you on the edge of your seat. |
| The Abyss: Special Edition | Ed Harris, Mary Elizabeth Mastrantonio, Michael Biehn, Leo Burmeister, Todd Graff, John Bedford Lloyd, Kimberly Scott | In this thrilling, underwater action-adventure from writer-director James Cameron (Titanic, Terminator 2: Judgment Day, Aliens), a civilian oil-rig crew is recruited to conduct a search-and-rescue effort when a nuclear submarine mysteriously sinks. One diver (Ed Harris) soon finds himself on a spectacular odyssey over 25,000 feet below the ocean's surface, where he confronts a mysterious force that has the power to change the world or destroy it. Includes both the Special Edition, with 28 minutes of additional footage, and the original theatrical version, along with the 80-minute documentary Under Pressure: Making The Abyss, and much more. |
| Absence Of The Good | Stephen Baldwin, Rob Knepper, Shawn Hurt, Allen Garfield, Silas Weir Mitchell, Tyne Daly | One cop. One killer. No clues. No time. After his son is shot to death at school, Detective Caleb Barnes (Stephen Baldwin) loses touch with his soul. When a series of seemingly unrelated murders plagues Salt Lake City, the detective hides his grief in search for the killer. Hampered by a lack of clues and his commander's unrelenting pressure, Caleb painstakingly unravels a tangled web, exposing a malignant family history of abuse and murder. |

The print project method is as follows:

```
C_LONGINT(vLprint_height;$vLheight;vLprinted_height)
C_STRING(31;vSprint_area)
PAGE SETUP([Film];"Print_List3")
GET PRINTABLE AREA(vLprint_height)
vLprinted_height:=0
ALL RECORDS([Film])

vSprint_area:="Header" //Printing of header area
$vLheight:=Print form([Film];"Print_List3";Form header)
$vLheight:=21 //Fixed height
vLprinted_height:=vLprinted_height+$vLheight

While(Not(End selection([Film])))
   vSprint_area:="Detail" `Printing of detail area
   $vLheight:=Print form([Film];"Print_List3";Form detail)
 `Detail calculation is carried out in the form method
   vLprinted_height:=vLprinted_height+$vLheight
   If(OK=0) `CANCEL has been carried out in the form method
      PAGE BREAK
      vLprinted_height:=0
      vSprint_area:="Header" `Reprinting of the header area
      $vLheight:=Print form([Film];"Print_List3";Form header)
      $vLheight:=21
      vLprinted_height:=vLprinted_height+$vLheight
      vSprint_area:="Detail"
      $vLheight:=Print form([Film];"Print_List3";Form detail)
      vLprinted_height:=vLprinted_height+$vLheight
   End if
   NEXT RECORD([Film])
End while
PAGE BREAK `Make sure that the last page is printed
```

The Print_List3 form method is as follows:

```
C_LONGINT($l;$t;$r;$b;$fixed_wdth;$exact_hght;$l1;$t1;$r1;$b1)
C_LONGINT($final_pos;$i)
C_LONGINT($detail_pos;$header_pos;$hght_to_print;$hght_remaining)

Case of
   :(vSprint_area="Detail") `Printing of detail underway
      OBJECT GET COORDINATES([Film]Actors;$l;$t;$r;$b)
      $fixed_wdth:=$r-$l  `Calculation of the Actors text field size
      $exact_hght:=$b-$t
      OBJECT GET BEST SIZE([Film]Actors;$wdth;$hght;$fixed_wdth)
 `Optimal size of the field according to its contents
      $movement:=$hght-$exact_hght

      OBJECT GET COORDINATES([Film]Summary;$l1;$t1;$r1;$b1)
      $fixed_wdth1:=$r1-$l1  `Calculation of the Summary text field size
      $exact_hght1:=$b1-$t1
      OBJECT GET BEST SIZE([Film]Summary;$wdth1;$hght1;$fixed_wdth1)
 `Optimal size of the field according to its contents
      $movement1:=$hght1-$exact_hght1
      If($movement1>$movement)
 `We determine the highest field
         $movement:=$movement1
      End if

      If($movement>0)
         $position:=Get print marker(Form detail)
         $final_pos:=$position+$movement
 `We move the Detail marker and those that follow it
         SET PRINT MARKER(Form detail;$final_pos;*)
 `Resizing of text areas
         OBJECT MOVE([Film]Actors;$l;$t;$r;$hght+$t;*)
         OBJECT MOVE([Film]Summary;$l1;$t1;$r1;$hght1+$t1;*)

 `Resizing of dividing lines
```

```
            OBJECT GET COORDINATES(*;"H1Line";$l;$t;$r;$b)
            OBJECT MOVE(*;"H1Line";$l;$final_pos-1;$r;$final_pos;*)
            For($i;1;4;1)
               OBJECT GET COORDINATES(*;"VLine"+String($i);$l;$t;$r;$b)
               OBJECT MOVE(*;"VLine"+String($i);$l;$t;$r;$final_pos;*)
            End for
         End if

   `Calculation of available space
         $detail_pos:=Get print marker(Form detail)
         $header_pos:=Get print marker(Form header)
         $hght_to_print:=$detail_pos-$header_pos
         $hght_remaining:=printing_height-vLprinted_height
         If($hght_remaining<$hght_to_print) `Insufficient height
            CANCEL `Move form to the next page
         End if
   End case
```

# SET PRINT OPTION

SET PRINT OPTION ( option ; value1 {; value2} )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ⇒ | Option number or PDF option code |
| value1 | Longint, Text | ⇒ | Value 1 of the option |
| value2 | Longint, Text | ⇒ | Value 2 of the option |

## Description

The **SET PRINT OPTION** command is used to modify, by programming, the value of a print option. Each option defined using this command is applied to the entire database and for the duration of the session as long as no other command that modifies print parameters (**PRINT SETTINGS**, **PRINT SELECTION** without the > parameter, etc.) is called. If a print job has been opened, the option is set for the job and cannot be modified as long as the job has not terminated.

The *option* parameter allows you to indicate the option to be modified. You can pass either one of the predefined constants of the "**Print Options**" theme, or a PDF option code (usable with the PDFCreator driver under Windows only).

Pass the new value(s) of the specified *option* in the *value1* and (optionally) *value2* parameters. The number and nature of the values to be passed depend on the type of option specified.

### Using an option number (constant)

The following table lists the options and their possible values:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Paper option | Longint | 1 | If you use only *value1*, it contains the name of the paper. If you use both parameters, *value1* contains the paper width and *value2* contains the paper height. The width and height are expressed in screen pixels. Use the **PRINT OPTION VALUES** command to get the name, height and width of all the paper formats offered by the printer. |
| Orientation option | Longint | 2 | *value1* only: 1=Portrait, 2=Landscape. If a different orientation option is used, **GET PRINT OPTION** returns 0 in *value1*.<br>**64-bit versions**: This option can be called within a print job, which means that you can switch from portrait to landscape, or vice versa, during the same print job. |
| Scale option | Longint | 3 | *value1* only: scale value in percentage. Be careful, some printers do not allow you to modify the scale. If you pass an invalid value, the property is reset to 100% at the time of printing. |
| Number of copies option | Longint | 4 | *value1* only: number of copies to be printed. |
| Paper source option | Longint | 5 | (Windows only) *value1* only: number corresponding to the index, in the array of trays returned by the **PRINT OPTION VALUES** command, of the paper tray to be used. This option can only be used under Windows. |
| Color option | Longint | 8 | (Windows only) *value1* only: code specifying the mode for handling color: 1=Black and white (monochrome), 2=Color.<br>**64-bit versions**: This option is not supported in 4D 64-bit versions (obsolete) |
| Destination option | Longint | 9 | *value1*: code specifying the type of print destination: 1=Printer, 2=(PC)/PS File (Mac), 3=PDF file, 5=Screen (OS X driver option).<br>If *value1* is different from 1 or 5, *value2* contains pathname for resulting document. This path will be used until another path is specified. If a file with the same name already exists at the destination location, it will be replaced. With **GET PRINT OPTION**, if the current value is not in the predefined list, *value1* contains -1 and the system variable OK is set to 1. If an error occurs, *value1* and the system variable OK are set to 0.<br>**Note**: Under Windows, you can set the printing destination to 3 (PDF File) when the PDF Creator driver has been installed. When the *(9;3;path)* values are passed, 4D automatically starts a "silent" PDF printing which takes into account any option codes that are passed (note that if you pass an empty string in *value2* or omit this parameter, a file saving dialog appears at the time of printing.) After printing, the current settings are restored. This simplifies control of printing PDFs for 4D and lets you write multi-platform code. When the *(9;3;path)* values are not passed, printing is not controlled by 4D and any PDF Creator option codes that were passed are ignored. |
| Double sided option | Longint | 11 | (Windows only) *value1*: 0=Single-sided or standard, 1=Double-sided. If *value1*=1, *value2* contains the binding: 0=Left binding (default value), 1=Top binding.<br>**Note:** This option can only be used under Windows. |
| Spooler document name option | Longint | 12 | *value1* only: name of the current print document, which appears in the list of spooler documents. The name defined by this statement will be used for all the print documents of the session for as long as a new name or an empty string is not passed. To use or restore standard operation (using the method name in the case of a method, the table name for a record, etc.), pass an empty string in *value1*. |
| Mac spool file format option | Longint | 13 | (Mac only) *value1* only: 0=print job in PDF mode (default value) 1=print job in PostScript mode.<br>**Notes:**<br>- This option has no effect under Windows.<br>- Under OS X, printing is done as a PDF by default. However, the PDF print driver does not support PICT pictures with encapsulated PostScript information — these pictures are generated, more particularly, by vectorial drawing software. To avoid this problem, this option lets you modify the print mode to use under OS X for the current session. Keep in mind that printing in PostScript mode can lead to undesired side effects.<br>**64-bit versions:** This option is not supported; it is replaced by the Generic PDF driver option of the **SET CURRENT PRINTER** command. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Hide printing progress option | Longint | 14 | *value1* only: 1=hide progress windows, 0=display progress windows (default). This option is particularly useful in the case of PDF printing under OS X.<br>**Note:** There is already a Printing progress option found in the Database Settings dialog box (Interface page). However, it is applied globally to the application and does not hide all the windows under OS X. |
| Page range option | Longint | 15 | *value1*=first page to print (default value is 1) and (optional) *value2*=number of the last page to print (default value -1 = end of document). |
| Legacy printing layer option | Longint | 16 | (4D 64-bit versions for Windows only) *value1* only: 1=select the GDI-based legacy printing layer for the subsequent printing jobs. 0=select the D2D printing layer (default).<br>**64-bit versions:** This selector is only supported on 4D 64-bit single-user applications on Windows; it is ignored on other platforms. It is mainly intended to allow legacy plug-ins to print inside 4D jobs in 4D 64-bit applications. |

Once set using this command, a print option is kept throughout the duration of the session for the entire 4D application. It will be used by the **PRINT SELECTION**, **PRINT RECORD**, **Print form**, **QR REPORT** and **WP PRINT** commands, as well as for all 4D printing, including that in Design mode.

**Notes:**

- It is indispensable to use the optional > parameter with the **PRINT SELECTION**, **PRINT RECORD** and **PAGE BREAK** commands in order to avoid resetting the print options that were set using the **SET PRINT OPTION** command.
- The **SET PRINT OPTION** command mainly supports PostScript printers. You can use this command with other types of printers, such as PCL or Ink, but in this case, it is possible that some options may not be available.

### Using a PDF option code (Windows)

In order to be able to use a PDF option code in the *option* parameter, you must have installed the PDFCreator driver in your 4D environment (for more information, refer to the **Integration of PDFCreator driver under Windows** section). Moreover, in order for option codes to be taken into account, you need to have enabled control of PDF printing for 4D using the following statement:

```
SET PRINT OPTION(Destination option;3;fileName)
```

Otherwise, option codes are ignored.

A PDFoption code is a Text type value consisting of two parts, *OptionType* and *OptionName*, combined together as "*OptionType:OptionName*". Here is the description of this code:

- *OptionType* Indicates whether you are specifying a native PDFCreator option or a 4D PDF administration option. Two values are accepted:
  - **PDFOptions** = native option
  - **PDFInfo** = internal option.
- *OptionName* Specifies the option to be set (depending on the *OptionType* value).
  - If *OptionType* = **PDFOptions**, you can pass one of several PDFCreator native options in *OptionName*. For example, the UseAutosave option affects the automatic backup. In order to be able to modify this option, pass "PDFOptions:UseAutosave" in the *option* parameter and the value to be used in the *value1* parameter. For a complete description of the PDFCreator native options, please refer to the documentation provided with the PDFCreator driver.
  - If *OptionType* = **PDFInfo**, you can pass one of the following specific selectors in *OptionName*:
    - **Reset print**: used to reset the internal waiting status in order, more particularly, to exit from an infinite loop. In this case, *value1* is not used.
    - **Reset standard options**: used to reset all the PDFCreator options to their default values. If printing is in progress, the default settings are applied after its completion. In this case, *value1* is not used.
    - **Start**: used to start or stop the PDFCreator spooler. Pass 0 in *value1* to stop it and 1 to start it.
    - **Reset options**: used to reset all the options modified since the beginning of the session using the **SET PRINT OPTION** command and the **PDFOptions** selector.
    - **Version**: used to read the current version number of the PDFCreator driver. This selector can only be used with the **GET PRINT OPTION** command. The number is returned in the *value1* parameter.
    - **Last error**: used to read the last error returned by the PDFCreator driver. This selector can only be used with the **GET PRINT OPTION** command. The error number is returned in the *value1* parameter.
    - **Print in progress**: used to find out if 4D is waiting for printing by PDFCreator. This selector can only be used with the **GET PRINT OPTION** command. The *value1* parameter returns 1 if 4D is waiting for PDFCreator and 0 otherwise.

- **Job count**: used to find out the number of jobs waiting in the printing queue. This selector can only be used with the **GET PRINT OPTION** command. The number of jobs is returned in the *value1* parameter.
- **Synchronous Mode**: used to set the synchronization mode betwen printing requests sent by 4D and the PDFCreator driver. Since 4D cannot get information about the current status of a print job that is in the printing queue, this option can be used to better control the execution of the jobs by only sending them when the status of the PDFCreator driver is "free". In this case, 4D is synchronized with the driver. Pass 0 in *value1* for 4D to send print requests immediately (default value) and 1 in order for 4D to be synchronized and to wait for the driver to have finished the job in progress before sending another one.

**Note:** After each printing, 4D automatically re-establishes the previous settings of the PDFCreator driver in order to avoid any interference with other programs using PDFCreator.

## Example 1

The following method enables the PDF driver so as to print all the records of the table at the C:¥Test¥Test_PDF_X location where X is the sequence number of the record:

```
SET CURRENT PRINTER(PDFCreator Printer Name)
 // Under Windows, select the virtual printer installed by PDFCreator
If(OK=1) // If PDFCreator is actually installed

   ALL RECORDS([Table_1])
   For($i;1;Records in selection([Table_1]))
      SET PRINT OPTION(Destination option;3;"C:¥¥Test¥¥Test_PDF_"+String($i))
 // Destination option 3 launches a PDFCreator print job
      PRINT RECORD([Table_1];*)
      NEXT RECORD([Table_1])
   End for
 // Resetting of the PDFCreator driver options
   SET PRINT OPTION("PDFInfo:Reset standard options";0)
End if
```

## Example 2

In 64-bit versions, the value of Orientation option can be modified within the same print job (special case). Note that the option must have been set before the **PAGE BREAK** command:

```
ALL RECORDS([People])
PRINT SETTINGS
If(OK=1)
   OPEN PRINTING JOB
   SET PRINT OPTION(Orientation option;1) //portrait
   Print form([People];"Vertical_Form")

   SET PRINT OPTION(Orientation option;2) //landscape
   PAGE BREAK //must be called imperatively AFTER the option
   Print form([People];"Horiz_Form")
   CLOSE PRINTING JOB
End if
```

## System variables and sets

The system variable OK is set to 1 if the command has been executed correctly; otherwise, it is set to 0.

If you pass an invalid option code (option not recognized by PDFCreator for example), OK is set to 0.

## Error management

If the value passed for an *option* is invalid or if it is not available on the printer, the command returns an error (that you can intercept using an error-handling method installed by the **ON ERR CALL** command) and the current value of the option remains unchanged.

# SET PRINT PREVIEW

SET PRINT PREVIEW ( preview )

| Parameter | Type | | Description |
|---|---|---|---|
| preview | Boolean | ⇒ | Preview on screen (TRUE), or No preview (FALSE) |

## Description

**SET PRINT PREVIEW** allows you to programmatically check or uncheck the Preview on Screen option of the Print dialog box. If you pass TRUE in *preview*, Preview on Screen will be checked, if you pass FALSE in *preview*, Preview on Screen will be unchecked. This setting is local to a process and does not affect the printing of other processes or users.

## Example

The following example turns on the Preview on Screen option to display the results of a query on screen, and then turns it off.

```
QUERY([Customers])
If(OK=1)
   SET PRINT PREVIEW(True)
   PRINT SELECTION([Customers] ;*)
   SET PRINT PREVIEW(False)
End if
```

# SET PRINTABLE MARGIN

SET PRINTABLE MARGIN ( left ; top ; right ; bottom )

| Parameter | Type | | Description |
|---|---|---|---|
| left | Longint | ➡ | Left margin |
| top | Longint | ➡ | Top margin |
| right | Longint | ➡ | Right margin |
| bottom | Longint | ➡ | Bottom margin |

## Description

The **SET PRINTABLE MARGIN** command sets the values of various printing margins by using the **Print form**, **PRINT SELECTION** and **PRINT RECORD** commands.

You can pass one of the following values in the *left*, *top*, *right* and *bottom* parameters:

- 0 = use paper margins
- -1 = use printer margins
- value > 0 = margin in pixels (remember that 1 pixel in 72 dpi represents approximately 0.4 mm)

The values of the *right* and *bottom* parameters relate to the right and bottom edges of the paper respectively.

**Note:** For more information regarding Printing management and terminology in 4D, refer to the **GET PRINTABLE MARGIN** command description.

By default, 4D bases its printouts on the printer margins. Once the **SET PRINTABLE MARGIN** command is executed, the modified parameters are retained in the same process for the entire session.

## Example 1

The following example gets the size of the dead margin:

```
SET PRINTABLE MARGIN(-1;-1;-1;-1) `Sets the printer margin
GET PRINTABLE MARGIN($l;$t;$r;$b)
 `$l, $t, $r and $b correspond to the dead margins of the sheet
```

## Example 2

The following example gets the paper size:

```
SET PRINTABLE MARGIN(0:0:0:0) `Sets the paper margin
GET PRINTABLE AREA($height;$width)
 `For size A4: $height=842 ; $width=595 pixels
```

# Subtotal

Subtotal ( data {; pageBreak} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| data | Field | ⇒ | Numeric field or variable to return subtotal |
| pageBreak | Longint | ⇒ | Break level for which to cause a page break |
| Function result | Real | ⇐ | Subtotal of data |

## Description

**Subtotal** returns the subtotal for *data* for the current or last break level. **Subtotal** works only when a sorted selection is being printed with **PRINT SELECTION** or when printing using Print in the Design environment. The *data* parameter must be of type real, integer, or long integer. Assign the result of the **Subtotal** function to a variable placed in the Break area of the form.

**Warning:** You **must** execute **BREAK LEVEL** and **ACCUMULATE** before every form report for which you want to do break processing and calculate subtotals. See discussion at the end of the description of this command.

The second, optional, argument to **Subtotal** is used to cause page breaks during printing. If *pageBreak* is 0, **Subtotal** does not issue a page break. If *pageBreak* equals 1, **Subtotal** issues a page break for each level 1 break. If *pageBreak* equals 2, **Subtotal** issues a page break for each level 1 and level 2 break, and so on.

**Tip:** If you execute **Subtotal** from within an output form displayed at the screen, an error will be generated, triggering an infinite loop of updates between the form and the error window. To get out of this loop, press Alt+Shift (Windows) or Option-Shift (Macintosh) when you click on the Abort button in the Error window (you may have to do so several times). This temporarily stops the updates for the form's window. Select another form as the output form so the error will occur again. Go back to the Design Environment and isolate the call to **Subtotal** into a test **Form event**= On Printing Break if you use the form both for display and printing.

## Example

The following example is a one-line object method in a Break area of a form (B0, the area above the B0 marker). The *vSalary* variable is placed in the Break area. The variable is assigned the subtotal of the Salary field for this break level. Break processing must have been activated beforehand using the **BREAK LEVEL** and **ACCUMULATE** commands.

```
Case of
   :(Form event=On Printing Break)
      vSalary:=Subtotal([Employees]Salary)
End case
```

For more information about designing forms with header and break areas, see the 4D Design Reference manual.

## Activating Break Processing in Form Reports

In order to generate reports with breaks, break processing in form reports can be activated by calling the **BREAK LEVEL** and **ACCUMULATE** commands.

You must execute both of these commands before printing a form report. The **Subtotal** function is still required in order to display values on a form. You must sort on at least as many levels as you need to break on.

When using **BREAK LEVEL** and **ACCUMULATE**, the process to print a report is typically like this:

1. Select the records to be printed.
2. Sort the records using **ORDER BY**. Sort on at least the same number of levels as breaks.
3. Execute **BREAK LEVEL** and **ACCUMULATE**.
4. Print the report using **PRINT SELECTION**.

The **Subtotal** function is necessary in order to display values on a form.

# Process (Communications)


- About semaphores
- About workers
- CALL PROCESS
- CALL WORKER
- CLEAR SEMAPHORE
- GET PROCESS VARIABLE
- KILL WORKER
- Semaphore
- SET PROCESS VARIABLE
- Test semaphore
- VARIABLE TO VARIABLE

# About semaphores

## What is a semaphore?

In a computer program, a semaphore is a tool used to protect actions that need to be performed by only one process or user at a time.

In 4D, the conventional need for using semaphores is for modifying an interprocess array: if one process is modifying the values of the array, another process must not be able to do the same thing at the same time. The developer uses a semaphore to indicate to a process that it can only perform its sequence of operations if no other process is already performing the same tasks. When a process encounters a semaphore, there are three possibilities:

- It immediately gets the right to pass
- It waits for its turn until it gets the right to pass
- It continues on its way, abandoning the idea of performing the tasks.

The semaphore therefore protects parts of the code. It allows only one process through at a time and blocks access until the process currently holding the right of use relinquishes this right by releasing the semaphore.

## Commands for working with semaphores

In 4D, you set a semaphore by calling the **Semaphore** function. To release a semaphore, you call the **CLEAR SEMAPHORE** command.

The **Semaphore** function has a very special behavior since it potentially performs two actions simultaneously:

- If the semaphore is already assigned, the function returns **True**
- If the semaphore is not assigned, the function assigns it to the process and returns **False** at the same time.

This double action performed by the same command ensures that no external operation can be inserted between the semaphore test and its assignment.

You can use the **Test semaphore** command to find out whether a semaphore is already assigned or not. This command is mainly used as part of long operations, such as the annual closing of accounts, where **Test semaphore** lets you control the interface to prevent access to certain operations such as the addition of accounting data.

## How to use semaphores

Semaphores should be used according to the following principles:

- A semaphore must be set and released in the same method,
- The execution of code protected by the semaphore must be as short as possible,
- The code must be timed by means of the *tickCount* parameter of the **Semaphore** function to wait for the release of the semaphore.

Here is typical code for using a semaphore:

```
While(Semaphore("MySemaphore";300))
   IDLE
End while
 // place code protected by semaphore here
CLEAR SEMAPHORE("MySemaphore")
```

A semaphore that is not released can block part of the database. Setting and releasing semaphores in the same method helps to eliminate this risk.

Minimizing the code protected by the semaphore increases the fluidity of the application and avoids the semaphore acting as a bottleneck.

Finally, using the optional *tickCount* parameter of the **Semaphore** command is essential for optimizing the wait for the semaphore to be released. Using this parameter, the commands works as follows:

- The process waits a maximum of the specified number of ticks (300 in the example) for the semaphore to be available, without the code execution moving on to the next line,
- If the semaphore is released before the end of this limit, it is immediately assigned to the process (**Semaphore** returns False) and code execution resumes,
- If the semaphore is not released before the end of this limit, then code execution resumes.

The command also prioritizes requests by establishing a queue. This way, the first process requesting a semaphore will be the first to get one.

Note that the wait time is set depending on the specifics of the application.

## Local or global semaphores

There are two types of semaphores in 4D: local semaphores and global semaphores.

- A local semaphore is accessible by all processes on the same workstation and only on the workstation. A local semaphore can be created by prefixing the name of the semaphore with a dollar sign ($). You use local semaphores to monitor operations among processes executing on the same workstation. For example, a local semaphore can be used to monitor access to an interprocess array shared by all the processes in your single-user database or on the workstation.

- A global semaphore is accessible to all users and all their processes. You use global semaphores to monitor operations among users of a multi-user database.

Global and local semaphores are identical in their logic. The difference resides in their scope.

In client-server mode, global semaphores are shared among all the processes running on all clients and servers. A local semaphore is only shared among the processes running on the machine where it has been created.

In 4D, global or local semaphores have the same scope because you are the only user. However, if your database is being used in both setups, make sure to use global or local semaphores depending on what you want to do.

**Note:** We recommend using local semaphores when you need a semaphore to manage a local aspect for a client of the application, such as the interface or an array of interprocess variables. If you use a global semaphores in this case, it would not only cause unnecessary network exchanges but could also affect other client machines unnecessarily. Using a local semaphore would avoid these undesirable side effects.

# About workers

## Overview

A **Worker process** is a simple and powerful way to exchange information between processes.  This feature is based upon an asynchronous messaging system that allows processes and forms to be called and asked to execute methods with parameters in their own context.

**Note:** A project method can also be executed with parameters in the context of any form using the **CALL FORM** command.

A worker can be "hired" by any process (using the **CALL WORKER** command) to execute project methods with parameters in their own context, thus allowing access to shared information.

This functionality addresses the following needs regarding 4D interprocess communication:

- Since they are supported by both cooperative and preemptive processes, they are the perfect solution for interprocess communication in preemptive processes (interprocess variables are not allowed in preemptive processes).
- They provide a simple alternative to semaphores, which can be cumbersome to set and complex to use (see **About semaphores**).

**Note:** Although they have been designed mainly for interprocess communication in the context of preemptive processes (64-bit versions only), **CALL WORKER** and **CALL FORM** are available in 32-bit versions and can be used with cooperative processes.

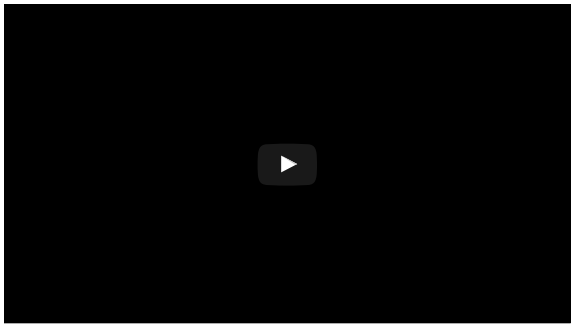
## Using workers

A **worker** is used to ask a process to execute project methods. A worker consists of:

- a unique name, also used to name its associated process
- an associated process, that may or may not exist at a given moment
- a message box
- a startup method (optional)

You ask a worker to execute a project method by calling the **CALL WORKER** command. The worker and its message box are created at first use; its associated process is also automatically launched at first use. If the worker process dies thereafter, the message box remains open and any new message in the box will start a new worker process.

The following animation illustrates this sequence:



Unlike the **New process** command, a worker process remains alive after the execution of the process method ends. This means that all method executions for the same worker will be run in the same process, which maintains all process state information (process variables, current record and current selection, etc.). Consequently, methods executed successively will access and thus share the same information, allowing communication between processes. The worker's message box handles successive calls asynchronously.

**CALL WORKER** encapsulates both the method name and command arguments in a message that is posted in the worker's message box. The worker process is then started, if it does not already exist, and asked to execute the message. This means that **CALL WORKER** will usually return before the method is actually executed (processing is asynchronous). For this reason, **CALL WORKER** does not return any value. If you need a worker to send information back to the process which called it (callback), you need to use **CALL WORKER** again to pass the information needed to the caller. Of course, in this case, the caller itself must be a worker.

It is not possible to use **CALL WORKER** to execute a method in a process created by the **New process** command. Only worker processes have a message box and can thus be called by **CALL WORKER**. Note that a process created by **New**

**process** can call a worker, but cannot be called back.

Worker processes can be created on 4D Server through stored procedures: for example, you can use the **Execute on server** command to execute a method that calls the **CALL WORKER** command.

A worker process is closed by a call to the **KILL WORKER** command, which empties the worker's message box and asks the associated process to stop processing messages and to terminate its current execution as soon as the current task is finished.

The startup method of a worker is the method used to create the worker (at first use). If **CALL WORKER** is called with an empty *method* parameter, then the startup method is automatically reused as method to execute.

The main process created by 4D when opening a database for user and application modes is a worker process and can be called using **CALL WORKER**. Note that the name of the main process may vary depending on the 4D localization language, but it always has the process number 1; as a result, it's more convenient to designate it by process number instead of process name when calling **CALL WORKER**.

### Identifying Worker processes

All worker processes, except the main process, have the process type Worker process (5) returned by the **PROCESS PROPERTIES** command.

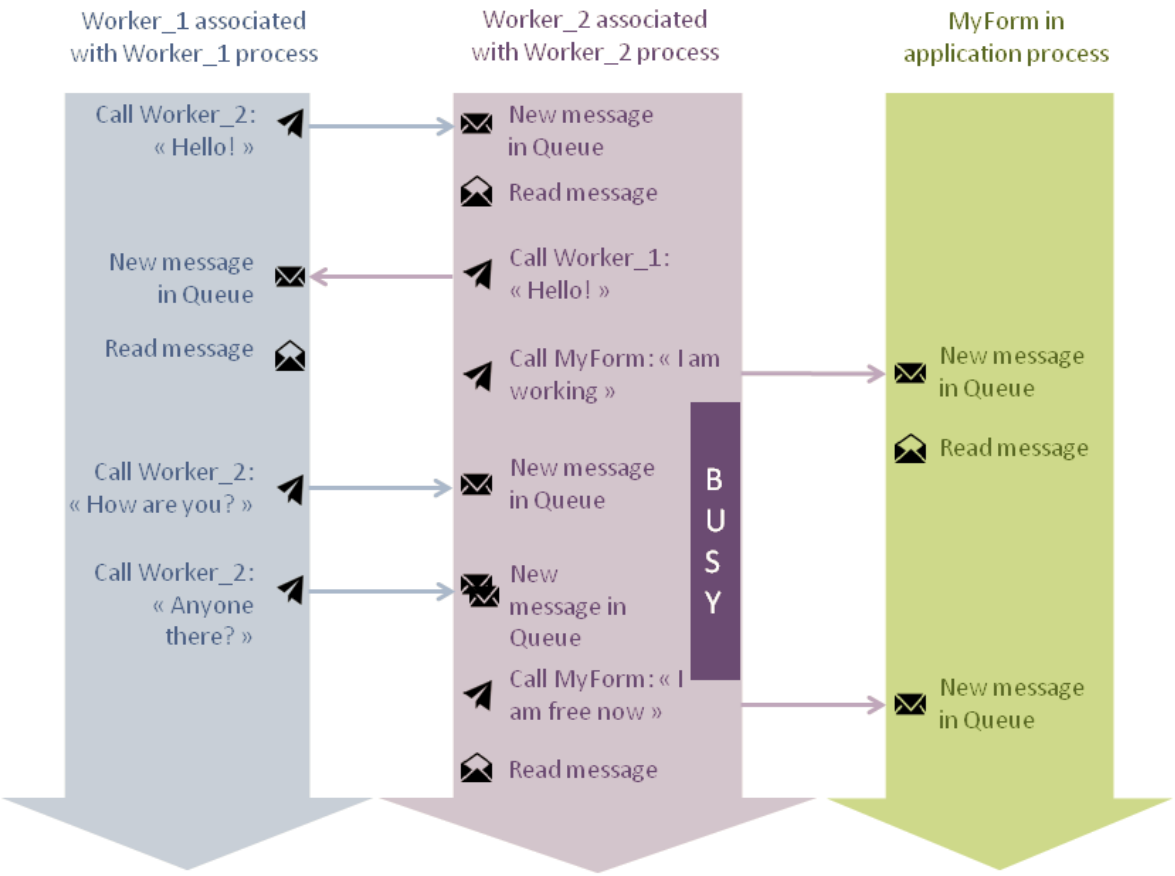
New icons identify worker processes in the Runtime Explorer and the 4D Server administration window:

| Process type | Icon |
|---|---|
| Preemptive worker process |  |
| Cooperative worker process |  |

**Note:** Other existing process icons have been updated in 4D v15 R5.

## Principles of use

The following graphic shows a communication sequence between two workers and a form. Information is exchanged by means of strings:




1. Worker_1 calls Worker_2 and passes "Hello!" as parameter.

2. Worker_2 gets the message and reads it. It calls Worker_1 back and passes "Hello!" as parameter.
3. Worker_2 then calls the MyForm form and passes "I am working" as parameter. It begins a time-consuming operation and its status becomes 'busy'.
4. Worker_1 calls Worker_2 twice but, thanks to the asynchronous messaging system, messages are simply queued. They are processed once Worker_2 is available - after Worker_2 has called MyForm.

# CALL PROCESS

CALL PROCESS ( process )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ⇒ | Process number |

## Description

**CALL PROCESS** calls the form displayed in the frontmost window of *process*.

**Important: CALL PROCESS** only works between processes running on the same machine.

If you call a process that does not exist, nothing happens.

If *process* (the target process) is not currently displaying a form, nothing happens. The form displayed in the target process receives an On Outside Call event. This event must be enabled for that form in the Design environment **Form Properties** window, and you must manage the event in the form method. If the event is not enabled or if it is not managed in the form method, nothing happens.

**Note:** The On Outside Call event modifies the entry context of the receiving input form. In particular, if a field was being edited, the On Data Change event is generated.

The caller process (the process from which **CALL PROCESS** is executed) does not "wait"— **CALL PROCESS** has an immediate effect. If necessary, you must write a waiting loop for a reply from the called process, using interprocess variables or using process variables (reserved for this purpose) that you can read and write between the two processes (using **GET PROCESS VARIABLE** and **SET PROCESS VARIABLE**).

To communicate between processes that do not display forms, use the **GET PROCESS VARIABLE** and **SET PROCESS VARIABLE** commands.

**CALL PROCESS** has the alternate syntax **CALL PROCESS**(-1).

In order not to slow down the execution of methods, 4D does not redraw interprocess variables each time they are modified. If you pass -1 instead of a process reference number in the *process* parameter, 4D does not call any process. Instead, it redraws all the interprocess variables currently displayed in all windows of any process running on the same machine.

## Example

See example for **On Exit database method**.

# CALL WORKER

CALL WORKER ( process ; method {; param}{; param2 ; ... ; paramN} )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Text, Longint | ⇒ | Name or number of worker process |
| method | Text | ⇒ | Name of project method to call |
| param | Expression | ⇒ | Parameter(s) passed to method |

## Description

The **CALL WORKER** command creates or calls the worker process whose name or ID you passed in *process*, and requests the execution of the *method* in its context with the optional *param* parameter(s).

The **CALL WORKER** command encapsulates the *param* parameters into a message and posts it in the worker's message box. For more information on worker processes, please refer to the **About workers** section.

In the *process* parameter, you can specify the worker using its process name or its process number:

- If you pass the number of a process that does not exist, or if the process specified was not created by **CALL WORKER** nor by 4D itself (such as the main application process), **CALL WORKER** does nothing.
- If you pass the name of a process that does not exist, a new worker process is created.

**Note:** The **main process**, created by 4D when a database is opened for the user interface and the application mode, is a worker process and can be called by **CALL WORKER**. However, since its name can vary depending on the 4D language, it is preferable to designate this process using its number (always 1) when you use **CALL WORKER**.

The worker process appears in the list of processes of the Runtime Explorer and is returned by the **PROCESS PROPERTIES** command when applied to this process.

In *method*, you pass the name of the project method to execute in the context of the worker process. You can pass an empty string; in this case, the worker executes the method that was originally used to start its process, if any (i.e., the startup method of the worker).

**Note:** It is not possible to pass an empty string in *method* when the command calls the main process (process number 1) since it was not started using a project method. As a result, ***CALL WORKER*** *(1;"")* does nothing.

You can also pass parameters to the *method* using one or more optional *param* parameters. You pass parameters the same way you would pass them to a subroutine (see the **Passing Parameters to Methods** section). Upon starting execution in the context of the process, the process method receives the parameter values in *$1*, *$2*, and so on. Remember that arrays cannot be passed as parameters to a method. Furthermore, in the context of the **CALL WORKER** command, the following additional considerations need to be taken into account:

- Pointers to tables or fields are allowed.
- Pointers to variables, particularly local and process variables, are not recommended since these variables may be undefined at the moment they are being accessed by the process method.
- If you pass an Object type parameter, 4D creates a copy of the object in the destination process if the worker is in a process different from the one calling the **CALL WORKER** command.

A worker process remains alive until the application is closed or the **KILL WORKER** command is explicitly called for it. To free up memory, do not forget to call this command once a worker process is no longer needed.

## Example

In a form, a button starts a computation: for example, statistics for the selected year. The button creates or calls a worker process that computes the data while the user can continue to work in the form.

The method of the button is:

```
 //call the worker vWorkerName with the parameter
C_LONGINT(vYear)
vYear:=2015 // could have been selected by the user in the form
CALL WORKER("myWorker";"workerMethod";vYear;Current form window)
```

The code of *workerMethod* is:

```
 // this is the method of the worker
 // it can be preemptive or cooperative
C_LONGINT($1) //gets the year
C_LONGINT($2) //gets the window reference
C_OBJECT(vStatResults) //to store statistical results
... //compute statistics
 //once finished, calls the form back with calculated values
 //vStatResults can display results in the form
CALL FORM($2;"displayStats";vStatResults)
```

# CLEAR SEMAPHORE

CLEAR SEMAPHORE ( semaphore )

| Parameter | Type | | Description |
|---|---|---|---|
| semaphore | String | ⇒ | Semaphore to clear |

## Description

**CLEAR SEMAPHORE** erases *semaphore* previously set by the **Semaphore** function.

As a rule, all semaphores that have been created should be cleared. If semaphores are not cleared, they remain in memory until the process that creates them ends. A process can only clear semaphores that it has created. If you try to clear a semaphore from within a process that did not create it, nothing happens.

## Example

See the example for **Semaphore**.

# GET PROCESS VARIABLE

GET PROCESS VARIABLE ( process ; srcVar ; dstVar {; srcVar2 ; dstVar2 ; ... ; srcVarN ; dstVarN} )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ⇨ | Source process number |
| srcVar | Variable | ⇨ | Source variable |
| dstVar | Variable | ⇦ | Destination variable |

## Description

The **GET PROCESS VARIABLE** command reads the *srcVar* process variables (*srvVar2*, etc.) from the source process whose number is passed in *process*, and returns their current values in the *dstVar* variables ( *dstVar2*, etc.) of the current process.

Each source variable can be a variable, an array or an array element. However, see the restrictions listed later in this section.

In each couple of *srcVar;dstVar* variables, the two variables must be of compatible types, otherwise the values you obtain may be meaningless.

The current process “peeks” the variables from the source process—the source process is not warned in any way that another process is reading the instance of its variables.

**4D Server:** Using 4D Client, you can read variables in a destination process executed on the server machine (stored procedure). To do so, put a minus sign before the process ID number in the *process* parameter.
“Intermachine” process communication, provided by the commands **GET PROCESS VARIABLE**, **SET PROCESS VARIABLE** and **VARIABLE TO VARIABLE**, is possible from client to server only. It is always a client process that reads or write the variables of a stored procedure.

**Tip:** If you do not know the ID number of the server process, you can still use the interprocess variables of the server. To do so, you can use any negative value in *process*. In other words, it is not necessary to know the ID number of the process to be able to use the **GET PROCESS VARIABLE** command with the interprocess variables of the server. This is useful when a stored procedure is launched using the **On Server Startup database method**. As clients machines do not automatically know the ID number of that process, any negative value can be passed in the *process* parameter.

### Restrictions

**GET PROCESS VARIABLE** does not accept local variables as source variables.

On the other hand, the destination variables can be interprocess, process or local variables. You “receive” the values only into variables, not into fields.

**GET PROCESS VARIABLE** accepts any type of source process or interprocess variable, except:

- Pointers
- Array of pointers
- Two-dimensional arrays

The source process must be a user process; it cannot be a kernel process. If the source process does not exist, this command has no effect.

**Note:** In interpreted mode, if a source variable does not exist, the undefined value is returned. You can detect this by using the **Type** function to test the corresponding destination variable.

## Example 1

This line of code reads the value of the text variable *vtCurStatus* from the process whose number is *$vlProcess*. It returns the value in the process variable *vtInfo* of the current process:

```
GET PROCESS VARIABLE($vlProcess;vtCurStatus;vtInfo)
```

## Example 2

This line of code does the same thing, but returns the value in the local variable *$vtInfo* for the method executing in the current process:

```
GET PROCESS VARIABLE($vlProcess;vtCurStatus;$vtInfo)
```

## Example 3

This line of code does the same thing, but returns the value in the variable *vtCurStatus* of the current process:

```
GET PROCESS VARIABLE($vlProcess;vtCurStatus;vtCurStatus)
```

**Note:** The first *vtCurStatus* designates the instance of the variable in the source process The second *vtCurStatus* designates the instance of the variable in the current process.

## Example 4

This example sequentially reads the elements of a process array from the process indicated by *$vlProcess*:

```
GET PROCESS VARIABLE($vlProcess;vl_IPCom_Array;$vlSize)
For($vlElem;1;$vlSize)
   GET PROCESS VARIABLE($vlProcess;at_IPCom_Array{$vlElem};$vtElem)
  ` Do something with $vtElem
End for
```

**Note:** In this example, the process variable *vl_IPCom_Array* contains the size of the array *at_IPCom_Array*, and must be maintained by the source process.

## Example 5

This example does the same thing as the previous one, but reads the array as a whole, instead of reading the elements sequentially:

```
GET PROCESS VARIABLE($vlProcess;at_IPCom_Array;$anArray)
For($vlElem;1;Size of array($anArray))
  ` Do something with $anArray{$vlElem}
End for
```

## Example 6

This example reads the source process instances of the variables *v1*,*v2*,*v3* and returns their values in the instance of the same variables for the current process:

```
GET PROCESS VARIABLE($vlProcess;v1;v1;v2;v2;v3;v3)
```

## Example 7

See the example for the **DRAG AND DROP PROPERTIES** command.

# KILL WORKER

KILL WORKER {{ process )}

| Parameter | Type | | Description |
|---|---|---|---|
| process | Text, Longint | ⇒ | Number or name of process to kill (kills current process if omitted) |

## Description

The **KILL WORKER** command posts a message to the worker process whose name or number you passed in *process*, asking it to ignore any pending messages and to terminate its execution as soon as the current task ends.

This command can only be used with worker processes. For more information, please refer to the **About workers** section.

In *process*, you pass either the name or number of the worker process whose execution needs to be terminated. If no worker with the specified process name or number exists, **KILL WORKER** does nothing.
If you do not pass any parameter, **KILL WORKER** applies to the currently running worker and is therefore equivalent to ***KILL WORKER*** *(Current process)*.

If **KILL WORKER** is applied to a worker that was not created explicitly using the **CALL WORKER** command (for example, the main application worker), it only asks this worker to empty its message box. Consequently, ***KILL WORKER*** *(1)* does nothing.

## Example

The following code (executed from a form, for example) triggers the termination of a worker:

```
CALL WORKER(vWorkerName;"theWorker";"end")
```

In the worker method (*theWorker*), you add some code to handle this situation:

```
 //theWorker method
C_TEXT($1) //param

Case of
   :($1="call") //the worker is called
      ... //do something
   :($1="end") //the worker is asked to kill itself
      KILL WORKER
End case
```

# Semaphore

Semaphore ( semaphore {; tickCount} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| semaphore | String | ⇒ | Semaphore to test and set |
| tickCount | Longint | ⇒ | Maximum waiting time |
| Function result | Boolean | ⇐ | Semaphore has been successfully set (FALSE) or Semaphore was already set (TRUE) |

## Description

A semaphore is a flag shared among workstations or among processes on the same workstation. A semaphore simply exists or does not exist. The methods that each user is running can test for the existence of a semaphore. A semaphore can only be removed by the client workstation or process that created it. By creating and testing semaphores, methods can communicate between workstations. You do not use semaphores to protect record access. This is automatically done by 4D and 4D Server. Use semaphores to prevent several users from performing the same operation at the same time.

The **Semaphore** function returns TRUE and does nothing if *semaphore* already exists. If *semaphore* does not exist, **Semaphore** creates it and returns FALSE. Only one user at a time can create a semaphore. If **Semaphore** returns FALSE, it means that the semaphore did not exist, but it also means that the semaphore has been set for the process in which the call has been made.

**Semaphore** returns FALSE if the semaphore was not set. It also returns FALSE if the semaphore is already set by the same process in which the call has been made.

A semaphore is limited to 255 characters, including prefix ($). If you pass a longer string, the semaphore will be tested with the truncated string. Keep in mind that semaphore names are case-sensitive in 4D (for example, the program considers that "MySemaphore" is different from "mysemaphore").

The optional parameter *tickCount* allows you to specify a waiting time (in ticks) if *semaphore* is already set. In this case, the function will wait either for the semaphore to be freed or the waiting time to expire before returning **True**.

There are two types of semaphores in 4D: local semaphores and global semaphores.

- A local semaphore is accessible by all processes on the same workstation and only on the workstation. A local semaphore can be created by prefixing the name of the semaphore with a dollar sign ($). You use local semaphores to monitor operations among processes executing on the same workstation. For example, a local semaphore can be used to monitor access to an interprocess array shared by all the processes in your single-user database or on the workstation.
- A global semaphore is accessible to all users and all their processes. You use global semaphores to monitor operations among users of a multi-user database.

Global and local semaphores are identical in their logic. The difference resides in their scope.

In client-server mode, global semaphores are shared among all the processes running on all clients and servers. A local semaphore is only shared among the processes running on the machine where it has been created.

In 4D, global or local semaphores have the same scope because you are the only user. However, if your database is being used in both setups, make sure to use global or local semaphores depending on what you want to do.

**Note:** We recommend using local semaphores when you need a semaphore to manage a local aspect for a client of the application, such as the interface or an array of interprocess variables. If you use a global semaphores in this case, it would not only cause unnecessary network exchanges but could also affect other client machines unnecessarily. Using a local semaphore would avoid these undesirable side effects.

## Example 1

Here is typical code for using a semaphore:

```
While(Semaphore("MySemaphore";300))
   IDLE
End while
 // place code protected by semaphore here
CLEAR SEMAPHORE("MySemaphore")
```

## Example 2

In this example, you want to prevent two users from doing a global update of the prices in a Products table. The following method uses semaphores to manage this:

```
If(Semaphore("UpdatePrices")) ` Try to create the semaphore
   ALERT("Another user is already updating prices. Retry later.")
Else
   DoUpdatePrices ` Update all the prices
   CLEAR SEMAPHORE("UpdatePrices")) ` Clear the semaphore
End if
```

## Example 3

The following example uses a local semaphore. In a database with several processes, you want to maintain a To Do list. You want to maintain the list in an interprocess array and not in a table. You use a semaphore to prevent simultaneous access. In this situation, you only need to use a local semaphore, because your To Do list is only for your use.

The interprocess array is initialized in the Startup method:

```
ARRAY TEXT(◊ToDoList;0) ` The To Do list is initially empty
```

Here is the method used for adding items to the To Do list:

```
 ` ADD TO DO LIST project method
 ` ADD TO DO LIST ( Text )
 ` ADD TO DO LIST ( To do list item )
C_TEXT($1)
If(Not(Semaphore("$AccessToDoList";300)))
 ` Wait 5 seconds if the semaphore already exists
   $vlElem:=Size of array(◊ToDoList)+1
   INSERT IN ARAY(◊ToDoList;$vlElem)
   ◊ToDoList{$vlElem}:=$1
   CLEAR SEMAPHORE("$AccessToDoList") ` Clear the semaphore
End if
```

You can call the above method from any process.

## Example 4

This method allows you to not execute a method when a semaphore is present; the method alerts the calling method with an error code and plain text.

Syntax:

```
$L_Error:=Semaphore_proof(->$T_Text_error)
```

```
 // Protective structure using semaphores
C_LONGINT($0)
C_POINTER($1) // error message

 // Start of method
C_LONGINT($L_MyError)
$L_MyError:=1

C_TEXT($T_Sema_local)
$T_Sema_local:="$tictac"

If(Semaphore($T_Sema_local;300))
 // We expected 300 ticks but the semaphore
 // was not released by the one that placed it:
 // we end up here
   $L_MyError:=-1

Else
```

```
    // This method is only run by one process at a time

    // We placed the semaphore as we entered
    // so we're the only ones that can remove it

    // Do something
      ...
    // Then finish by removing the semaphore
      CLEAR SEMAPHORE($T_Sema_local)
End if

C_TEXT($T_Message)
If($L_MyError=-1)
   $T_Message:="The semaphore "+$T_Sema_local+" has blocked access to the rest of the code"
Else
   $T_Message:="OK"
End if

$0:=$L_MyError
$1->:=$T_Message  // The calling method receives an error code and an explanation in plain text
```

# SET PROCESS VARIABLE

SET PROCESS VARIABLE ( process ; dstVar ; expr {; dstVar2 ; expr2 ; ... ; dstVarN ; exprN} )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ➡ | Destination process number |
| dstVar | Variable | ➡ | Destination variable |
| expr | Variable | ➡ | Source expression (or source variable) |

## Description

The **SET PROCESS VARIABLE** command writes the *dstVar* process variables (*dstVar2*, etc.) of the destination process whose number is passed in *process* using the values passed in *expr1* (*expr2*, etc.).

Each destination variable can be a variable or an array element. However, see the restrictions listed later in this section.

For each couple of *dstVar;expr* variables, the expression must be of a type compatible with the destination variable, otherwise you may end up with a meaningless value in the variable. In interpreted mode, if a destination variable does not exist, it is created and assigned with the expression.

The current process “pokes” the variables of the destination process—the destination process is not warned in any way that another process is writing the instance of its variables.

**4D Server:** Using 4D Client, you can write variables in a destination process executed on the server machine (stored procedure). To do so, put a minus sign before the process ID number in the *process* parameter.
“Intermachine” process communication, provided by the commands **SET PROCESS VARIABLE**, **GET PROCESS VARIABLE** and **VARIABLE TO VARIABLE**, is possible from client to server only. It is always a client process that reads or write the variables of a stored procedure.

**Tip:** If you do not know the ID number of the server process, you can still use the interprocess variables of the server. To do so, use any negative value in *process*. In other words, it is not necessary to know the ID number of the process to be able to use the **SET PROCESS VARIABLE** command with the interprocess variables of the server. This is useful when a stored procedure is launched using the **On Server Startup database method**. As client machines do not automatically know the ID number of that process, any negative value can be passed in the *process* parameter.

### Restrictions

**SET PROCESS VARIABLE** does not accept local variables as destination variables.

**SET PROCESS VARIABLE** accepts any type of destination process or interprocess variable, except:

- Pointers
- Arrays of any type. To write an array as a whole from one process to another one, use the command **VARIABLE TO VARIABLE**. Note, however, that **SET PROCESS VARIABLE** allows you to write the element of an array.
- You cannot write the element of an array of pointers or the element of a two-dimensional array.

The destination process must be a user process; it cannot be a kernel process. If the destination process does not exist, an error is generated. You can catch this error using an error-handling method installed with **ON ERR CALL**.

## Example 1

This line of code sets (to the empty string) the text variable *vtCurStatus* of the process whose number is *$vlProcess*:

```
SET PROCESS VARIABLE($vlProcess;vtCurStatus;"")
```

## Example 2

This line of code sets the text variable *vtCurStatus* of the process whose number is *$vlProcess* to the value of the variable *$vtInfo* from the executing method in the current process:

```
SET PROCESS VARIABLE($vlProcess;vtCurStatus;$vtInfo)
```

## Example 3

This line of code sets the text variable *vtCurStatus* of the process whose number is *$vlProcess* to the value of the same variable in the current process:

```
SET PROCESS VARIABLE($vlProcess;vtCurStatus;vtCurStatus)
```

**Note:** The first *vtCurStatus* designates the instance of the variable in the destination process. The second *vtCurStatus* designates the instance of the variable in the current process.

## Example 4

This example sequentially sets to uppercase all elements of a process array from the process indicated by *$vlProcess*:

```
GET PROCESS VARIABLE($vlProcess;vl_IPCom_Array;$vlSize)
For($vlElem;1;$vlSize)
   GET PROCESS VARIABLE($vlProcess;at_IPCom_Array{$vlElem};$vtElem)
   SET PROCESS VARIABLE($vlProcess;at_IPCom_Array{$vlElem};Uppercase($vtElem))
End for
```

**Note:** In this example, the process variable *vl_IPCom_Array* contains the size of the array *at_IPCom_Array* and must be maintained by the source/destination process.

## Example 5

This example writes the destination process instance of the variables *v1*, *v2* and *v3* using the instance of the same variables from the current process:

```
SET PROCESS VARIABLE($vlProcess;v1;v1;v2;v2;v3;v3)
```

# Test semaphore

Test semaphore ( semaphore ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| semaphore | String | ⇒ | Name of the semaphore to test |
| Function result | Boolean | ⇐ | True = the semaphore exists, False = the semaphore doesn't exist |

## Description

The **Test semaphore** command tests for the existence of a semaphore.

The difference between the **Semaphore** function and the **Test semaphore** function is that **Test semaphore** doesn't create the *semaphore* if it doesn't exist. If the *semaphore* exists, the function returns **True**. Otherwise, it returns **False**.

## Example

The following example allows you to know the state of a process (in our case, while modifying the code) without modifying *semaphore*:

```
$Win:=Open window(x1;x2;y1;y2;-Palette window)
Repeat
   If(Test semaphore("Encrypting code"))
      POSITION MESSAGE($x3;$y3)
      MESSAGE("Encrypting code being modified.")
   Else
      POSITION MESSAGE($x3;$y3)
      MESSAGE("Modification of the encrypting code authorized.")
   End if
Until(StopInfo)
CLOSE WINDOW
```

# VARIABLE TO VARIABLE

VARIABLE TO VARIABLE ( process ; dstVar ; srcVar {; dstVar2 ; srcVar2 ; ... ; dstVarN ; srcVarN} )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ➡ | Destination process number |
| dstVar | Variable | ➡ | Destination variable |
| srcVar | Variable | ➡ | Source variable |

## Description

The **VARIABLE TO VARIABLE** command writes the *dstVar* process variables (*dstVar2*, etc.) of the destination process whose number is passed in *process* using the values of the variables *srcVar1 srcVar2*, etc.

**VARIABLE TO VARIABLE** has the same action as **SET PROCESS VARIABLE**, with the following differences:

- You pass source expressions to **SET PROCESS VARIABLE**, and therefore cannot pass an array as a whole. You must exclusively pass source variables to **VARIABLE TO VARIABLE**, and therefore can pass an array as a whole.
- Each destination variable of **SET PROCESS VARIABLE** can be a variable or an array element, but cannot be an array as a whole. Each destination variable of **VARIABLE TO VARIABLE** can be a variable or an array or an array element.

**4D Server:** "Intermachine" process communication, provided by the commands **GET PROCESS VARIABLE**, **SET PROCESS VARIABLE** and **VARIABLE TO VARIABLE**, is possible from client to server only. It is always a client process that reads or write the variables of a stored procedure.

For each couple of *dstVar;expr* variables, the source variable must be of a type compatible with the destination variable, otherwise you may end up with a meaningless value in the variable. In interpreted mode, if a destination variable does not exist, it is created and assigned with the type and value of the source variable.

The current process "pokes" the variables of the destination process—the destination process is not warned in any way that another process is writing the instance of its variables.

### Restrictions

**VARIABLE TO VARIABLE** does not accept local variables as destination variables.

**VARIABLE TO VARIABLE** accepts any type of destination process or interprocess variables except:

- Pointers
- Array of pointers
- Two-dimensional arrays

The destination process must be a user process; it cannot be a kernel process. If the destination process does not exist, an error is generated. You can catch this error using an error-handling method installed with **ON ERR CALL**.

## Example

The following example reads a process array from the process indicated by *$vlProcess*, sequentially sets the elements to uppercase and then writes back the array as a whole:

```
GET PROCESS VARIABLE($vlProcess;at_IPCom_Array;$anArray)
For($vlElem;1;Size of array($anArray))
   $anArray{$vlElem}:=Uppercase($anArray{$vlElem})
End for
VARIABLE TO VARIABLE($vlProcess;at_IPCom_Array;$anArray)
```

# Process (User Interface)

- BRING TO FRONT
- Frontmost process
- HIDE PROCESS
- SHOW PROCESS

# BRING TO FRONT

BRING TO FRONT ( process )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ⇨ | Process number of the process to pass to the frontmost level |

## Description

**BRING TO FRONT** brings all the windows belonging to *process* to the front. If the process is already the frontmost process, the command does nothing. If the process is hidden, you must use **SHOW PROCESS** to display the process, otherwise **BRING TO FRONT** has no effect.

The Main and Design processes can be brought to the front using this command.

**Note:** When the process contains several windows and you want to pass a specific one to the front, it is preferable to use, for example, the **SET WINDOW RECT** command.

## Example

The following example is a method that can be executed from a menu. It checks to see if *◊vlAddCust_PID* is the frontmost process. If not, the method brings it to the front:

```
If(Frontmost process#◊vlAddCust_PID)
   BRING TO FRONT(◊vlAddCust_PID)
End if
```

# Frontmost process

Frontmost process {( * )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | Process number for first non-floating window |
| Function result | Integer | ⇐ | Number of the process whose windows are in the front |

## Description

**Frontmost process** returns the number of the process whose window (or windows) are in the front.

When you have one or more floating windows open, there are two window layers:

- Regular windows
- Floating windows

If the **Frontmost process** function is used from within a floating window form method or object method, the function returns the process reference number of the frontmost floating window in the floating window layer. If you specify the optional * parameter, the function returns the process reference number of the frontmost active window in the regular window layer.

## Example

See the example for **BRING TO FRONT**.

# HIDE PROCESS

HIDE PROCESS ( process )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ⇨ | Process number or process to be hidden |

## Description

**HIDE PROCESS** hides all windows that belong to *process*. All interface elements of *process* are hidden until the next **SHOW PROCESS**. The menu bar of the process is also hidden. This means that opening a window while the process is hidden does not make the screen redraw or display. If the process is already hidden, the command has no effect.

The only exception to this rule is the Debugger window. If the Debugger window is displayed when *process* is a hidden process, *process* is displayed and becomes the frontmost process.

If you do not want a *process* to be displayed when it is created, **HIDE PROCESS** should be the first command in the process method. The Main Process and Cache Manager processes cannot be hidden using this command.

Even though a process may be hidden, the process is still executing.

## Example

The following example hides all the windows belonging to the current process:

```
HIDE PROCESS(Current process)
```

# SHOW PROCESS

SHOW PROCESS ( process )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ➡ | Process number of process to be shown |

## Description

**SHOW PROCESS** displays all the windows belonging to *process*. This command does not bring the windows of *process* to the frontmost level. To do this, use the **BRING TO FRONT** command.

If the process was already displayed, the command has no effect.

## Example

The following example displays a process called Customers, if it has been previously hidden. The process reference to the Customers process is stored in the interprocess variable ◊*Customers*:

```
SHOW PROCESS(◊Customers)
```

# Processes

- Processes
- Preemptive 4D processes
- Count tasks
- Count user processes
- Count users
- Current process
- Current process name
- DELAY PROCESS
- EXECUTE ON CLIENT
- Execute on server
- GET REGISTERED CLIENTS
- New process
- PAUSE PROCESS
- Process aborted
- Process number
- PROCESS PROPERTIES
- Process state
- REGISTER CLIENT
- RESUME PROCESS
- UNREGISTER CLIENT

# Processes

Multi-tasking in 4D is the ability to have distinct database operations that are executed simultaneously. These operations are called processes.

Multiple processes are like multiple users on the same computer, each working on his or her own task. This essentially means that each method can be executed as a distinct database task.

This section covers the following topics:

- Creating and clearing processes
- Elements of a process
- User processes
- Processes created by 4D
- Local and global processes
- Record locking between processes

**Note**: This section does not cover stored procedures. See the **Stored Procedures** section in the 4D Server Reference manual.

## Creating and Clearing Processes

There are several ways to create a new process:

- Execute a method in the Design environment after checking the **New Process** check box in the **Execute Method** dialog box. The method chosen in the Execute Method dialog box is the process method.
- Processes can be run by choosing menu commands. In the **Menu Bar editor**, select the menu command and click the **Start a New Process** check box. The method associated with the menu command is the process method.
- Use the **New process** function. The method passed as a parameter to the **New process** function is the process method.
- Use the **Execute on server** function in order to create a stored procedure on the server. The method passed as a parameter of the function is the process method.
- Use the **CALL WORKER** command. If the worker process does not already exist, it is created.

A process can be cleared under the following conditions. The first two conditions are automatic:

- When the process method finishes executing
- When the user quits from the database
- If you stop the process procedurally or use the Abort button in the Debugger
- If you choose **Abort** in the Runtime Explorer
- If you call the **KILL WORKER** command (to delete a worker process only).

A process can create another process. Processes are not organized hierarchically—all processes are equal, regardless of the process from which they have been created. Once the “parent” process creates a “child” process, the child process will continue regardless of whether or not the parent process is still executing.

## Elements of a Process

Each process contains specific elements. There are three types of distinctly different elements in a process:

- **Interface elements**: Elements that are necessary to display a process.
- **Data elements**: Information that is related to the data in the database.
- **Language elements**: Elements that are used procedurally or are that are important for developing your own application.

### Interface Elements

Interface elements consist of the following:

- **Menu bar**: Each process can have its own current menu bar. The menu bar of the frontmost process is the current menu bar for the database.
- **One or more windows**: Each process can have more than one window open simultaneously. On the other hand, some processes have no windows at all.
- **One active (frontmost) window**: Even though a process can have several windows open simultaneously, each process has only one active window. To have more than one active window, you must start more than one process.

**Notes:**

- Processes do not include menu bars by default which means that the standard **Edit** menu shortcuts (in particular, cut/copy/paste) are not availalbe in process windows. When you call dialog boxes or 4D editors (form editor, query editor, **Request**, etc.) from a process, if you want for the user to be able to benefit from keyboard shortcuts like copy/paste, you need to make sure that the equivalent of an **Edit** menu is installed in the process.
- Preemptive processes and processes that are executed on the server (stored procedures) must not contain elements of the interface.

### Data Elements

Data elements refer to the data used by the database. The data elements are:

- **Current selection per table**: Each process has a separate current selection. One table can have a different current selection in different processes.
- **Current record per table**: Each table can have a different current record in each process.

**Note**: This description of the data elements is valid if your processes are global in scope. By default, all processes are global. See the "**Global and Local Processes**" section below.

### Language Elements

The language elements of a process are the elements related to programming in 4D.

- **Variables**: Every process has its own process variables. See **Variables** for more information. Process variables are recognized only within the domain of their native process.
- **Default table**: Each process has its own default table. However, note that the **DEFAULT TABLE** command is only a typing convention for programming.
- **Input and Output forms**: Default input and output forms can be set procedurally for each table in each process.
- **Process sets**: Each process has its own process sets. *LockedSet* is a process set. Process sets are cleared as soon as the process method ends.
- **On Error Call per process**: Each process has its own error-handling method.
- **Debugger window**: Each process can have its own Debugger window.

## User Processes

User processes are processes that you create to perform certain tasks. They share processing time with the kernel processes. As an example, Web connection processes are user processes.

The 4D application also creates processes for its own needs. The following processes are created and managed by 4D:

- **Main process**: The main process manages the display windows of the user interface.
- **Design process**: The Design process manages the windows and editors of the Design environment. There is no Design process in a compiled database that does not contain interpreted code.
- **Web Server process**: The Web Server process runs when the database is published on the Web. See the **Web server configuration and connection management** section for more information.
- **Cache Manager process**: The Cache Manager process manages disk I/ O for the database. This process is created as soon as 4D or 4D Server are run.
- **Indexing process**: The Indexing process manages the indexing of fields in a database as a separate process. This process is created when an index for a field is built or deleted.
- **On Event Manager process**: This process is created when an event-handling method is installed by the **ON EVENT CALL** command. It executes the event method installed by **ON EVENT CALL** whenever there is an event. The event method is the process method for this process. This process executes continuously, even if no method is executing. Event handling also occurs in the Design environment.

## Cooperative and preemptive processes

Starting with 4D v15 R5 64 bits, 4D allows you to create preemptive user processes in compiled mode. In previous versions, only cooperative user processes were available.

When run in *preemptive* mode, a process is dedicated to a CPU. Process management is then delegated to the system, which can allocate each CPU separately on a multi-core machine. When run in *cooperative* mode, all processes are managed by the parent application thread and share the same CPU, even on a multi-core machine.

As a result, in preemptive mode, overall performance of the application is improved, especially on multi-core machines, since multiple processes (*threads*) can *truly* run simultaneously. However, actual gains depend on the operations being executed. In return, since each thread is independent from the others in preemptive mode, and not managed directly by the application, there are specific constraints applied to methods that you want to be compliant with preemptive use. Additionally, preemptive execution is only available in certain specific contexts.

Management of preemptive processes is covered in the **Preemptive 4D processes** section.

## Global and Local Processes

Processes can be either global or local in scope. By default, all processes are global.

Global processes can perform any operation, including accessing and manipulating data. In most cases, you will want to use global processes.

Local processes should be used only for operations that do not access data. For example, you can use a local process to run an event-handling method or to control interface elements such as floating windows.

You specify that a process is local in scope through its name. The name of local process must start with a dollar sign (`$`).

**Warning**: If you attempt to access data from a local process, you access it though the main process, risking conflicts with operations performed within that process.

**4D Server**: Using local processes on the Client side for operations that do not require data access reserves more processing time for server-intensive tasks.

## Record Locking Between Processes

A record is locked when another process has successfully loaded the record for modification. A locked record can be loaded by another process, but cannot be modified. The record is unlocked only in the process in which the record is being modified. A table must be in read/write mode for a record to be loaded unlocked. For more information, refer to the **Record Locking** section.

# Preemptive 4D processes

4D Developer Edition 64-bit for Windows and OS X offers a powerful feature allowing you to **execute 4D code in preemptive processes**. Thanks to this new feature, your 4D compiled applications will be able to take full advantage of multi-core computers so that their execution will be faster and they can support more connected users.

## What is a preemptive process?

When run in *preemptive* mode, a process is dedicated to a CPU. Process management is then delegated to the system, which can allocate each CPU separately on a multi-core machine.

When run in *cooperative* mode (the only mode available in 4D until 4D v15 R5), all processes are managed by the parent application thread and share the same CPU, even on a multi-core machine.

As a result, in preemptive mode, overall performance of the application is improved, especially on multi-core machines, since multiple processes (*threads*) can *truly* run simultaneously. However, actual gains depend on the operations being executed.

In return, since each thread is independent from the others in preemptive mode, and not managed directly by the application, there are specific constraints applied to methods that you want to be compliant with preemptive use. Additionally, preemptive execution is only available in certain specific contexts.

## Availability of preemptive mode

The use of preemptive mode is available in **4D 64-bit versions** only. The following execution contexts are currently supported:

| | Preemptive execution |
|---|---|
| 4D Server | X |
| 4D remote | - |
| 4D single-user | X |
| Compiled mode | X |
| Interpreted mode | - |

If the execution context supports preemptive mode and if the method is "thread-safe", a new 4D process launched using the **New process** or **CALL WORKER** commands, or the "Run method" menu item, will be executed in a preemptive thread.

Otherwise, if you call **New process** or **CALL WORKER** from an execution context that is not supported (for example on a remote 4D machine), the process is always cooperative.

**Note:** You can run a process in preemptive mode from a 4D remote by starting a stored procedure on the server with the language, for example using **Execute on server**.

## Thread-safe vs thread-unsafe

4D code can only be run in a preemptive thread when certain specific conditions are met. Each part of the code being executed (commands, methods, variables, etc.) must be compliant with preemptive use. Elements that can be run in preemptive threads are called **thread-safe** and those that cannot be run in preemptive threads are called **thread-unsafe**.

**Note:** Since a thread is handled independently starting from the parent process method, the entire call chain must not include any thread-unsafe code; otherwise, preemptive execution will not be possible. This point is discussed in the **When is a process started preemptively?** paragraph.

The "thread safety" property of each element depends on the element itself:

- 4D commands: thread safety is an internal property. In the *Language Reference*, thread-safe commands are identified by the  icon. A large part of 4D commands can run in preemptive mode.
- Project methods: conditions for being thread-safe are listed in the **Writing a thread-safe method** paragraph.

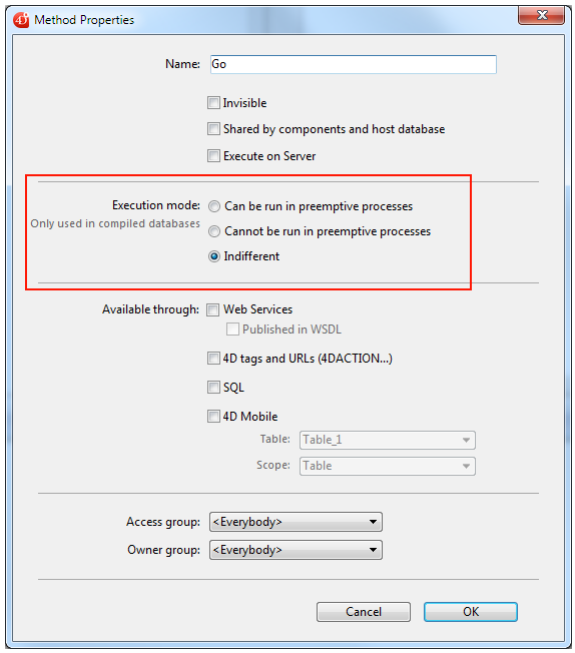
Basically, code to be run in preemptive threads cannot call parts with external interactions, such as plug-in code or interprocess variables. Accessing data, however, is allowed since the 4D data server supports preemptive execution.

## Declaration of preemptive execution mode

By default, 4D executes all the project methods of your application in cooperative mode. If you want to benefit from the preemptive mode feature, the first step consists in explicitly declaring all methods that you want to be started in preemptive mode whenever possible -- that is, methods that you consider capable of being run in a preemptive process. The compiler will check that these methods are actually thread-safe (see for more information). You can also disallow preemptive mode for some methods, if necessary.

Keep in mind that declaring a method "capable" of preemptive use makes it eligible for preemptive execution but does not guarantee that it will actually be executed in preemptive mode at runtime. Starting a process in preemptive mode results from an evaluation performed by 4D regarding the properties of all the methods in the call chain of the process (for more information, see the paragraph).

To declare your method eligible for use in preemptive mode, you need to use the **Execution mode** declaration option in the Method Properties dialog box:




The following options are provided:

- **Can be run in preemptive processes**: By checking this option, you declare that the method is capable of being run in a preemptive process and therefore should be run in preemptive mode whenever possible. The "preemptive" property of the method is set to "capable".
  When this option is checked, the 4D compiler will verify that the method is actually capable and will return errors if this is not the case -- for example, if it directly or indirectly calls commands or methods that cannot be run in preemptive mode (the entire call chain is parsed but errors are only reported to the first sublevel). You can then edit the method so that it becomes thread-safe, or select another option.
  If the method's preemptive capability is approved, it is tagged "thread-safe" internally and will be executed in preemptive mode whenever the required conditions are met. This property defines its eligibility for preemptive mode but does not guarantee that the method will actually be run in preemptive mode, since this execution mode requires a specific context (see **When is a process started preemptively?**).

- **Cannot be run in preemptive processes**: By checking this option, you declare that the method must never be run in preemptive mode, and therefore must always be run in cooperative mode, like in previous 4D versions. The "preemptive" property of the method is set to "incapable".
  When this option is checked, the 4D compiler will not verify the ability of the method to run preemptively; it is automatically tagged "thread-unsafe" internally (even if it is theoretically capable). When called at runtime, this method will "contaminate" any other methods in the same thread, thus forcing this thread to be executed in cooperative mode, even if the other methods are thread-safe.

- **Indifferent** (default): By checking this option, you declare that you do not want to handle the preemptive property for the method. The "preemptive" property of the method is set to "indifferent".
  When this option is checked, the 4D compiler will evaluate the preemptive capability of the method and will tag it internally as "thread-safe" or "thread-unsafe". No error related to preemptive execution is returned. If the method is evaluated as thread-safe, at runtime it will not prevent preemptive thread execution when called in a preemptive

context. Conversely, if the method is evaluated "thread-unsafe", at runtime it will prevent any preemptive thread execution when called.
Note that with this option, whatever the internal thread safety evaluation, the method will always be executed in cooperative mode when called directly by 4D as the first parent method (for example through the **New process** command). If tagged "thread-safe" internally, it is only taken into account when called from other methods inside a call chain.

**Note:** A component method declared as "Shared with components and host databases" must also be declared "capable" in order to be run in a preemptive thread by the host database.

When exporting the method code using, for example, **METHOD GET CODE**, the "preemptive" property is exported in the "%attributes" comment with a value of "capable" or "incapable" (the property is not available if the option is "Indifferent"). The **METHOD GET ATTRIBUTES** and **METHOD SET ATTRIBUTES** commands also get or set the "preemptive" attribute with a value of "indifferent", "capable", or "incapable".

The following table summarizes the effects of the preemptive mode declaration options:

| Option | Preemptive property value (interpreted) | Compiler action | Internal tag (compiled) | Execution mode if call chain is thread-safe |
|---|---|---|---|---|
| Can be run in preemptive processes | capable | Check capability and return errors if incapable | thread-safe | Preemptive |
| Cannot be run in preemptive processes | incapable | No evaluation | thread-unsafe | Cooperative |
| Indifferent | indifferent | Evaluationbut no errors returned | thread-safe or thread-unsafe | If thread-safe: preemptive; if thread-unsafe: cooperative; if called directly: cooperative |

## When is a process started preemptively?

**Reminder:** Preemptive execution is only available in compiled mode.

In compiled mode, when starting a process created by either **New process** or **CALL WORKER** methods, 4D reads the preemptive property of the process method (also named *parent* method) and executes the process in preemptive or cooperative mode, depending on this property:

- If the process method is thread-safe (validated during compilation), the process is executed in a preemptive thread.
- If the process method is thread-unsafe, the process is run in a cooperative thread.
- If the preemptive property of the process method was set to "indifferent", by compatibility the process is run in a cooperative thread (even if the method is actually capable of preemptive use). Note however that this compatibility feature is only applied when the method is used as a process method: a method declared "indifferent" but internally tagged "thread-safe" by the compiler can be called preemptively by another method (see below).

The actual thread-safe property depends on the call chain. If a method with the property declared as "capable" calls a thread-unsafe method at either of its sublevels, a compilation error will be returned: if a single method in the entire call chain is thread-unsafe, it will "contaminate" all other methods and preemptive execution will be rejected by the compiler. A preemptive thread can be created only when the entire chain is thread-safe and the process method has been declared "Can be run in preemptive process".
On the other hand, the same thread-safe method may be executed in a preemptive thread when it is in one call chain, and in a cooperative thread when it is in another call chain.

For example, consider the following project methods:

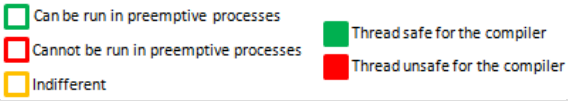
```
 //MyDialog project method
 //contains interface calls: will be internally thread unsafe
$win:=Open form window("tools";Palette form window)
DIALOG("tools")
```

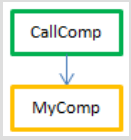
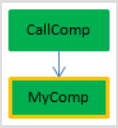
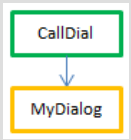
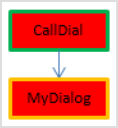
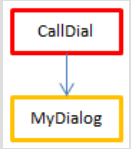
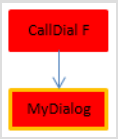
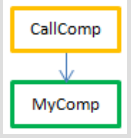
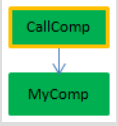
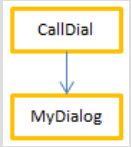
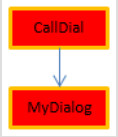
```
 //MyComp project method
 //contains simple computing: will be internally is thread safe
C_LONGINT($0;$1)
$0:=$1*2
```

```
 //CallDial project method
C_TEXT($vName)
MyDialog
```

```
 //CallComp project method
C_LONGINT($vAge)
MyCom($vAge)
```

Executing a method in preemptive mode will depend on it "execution" property and the call chain. The following table illustrates these various situations:




| Declaration and call chain | Compilation | Resulting thread safety | Execution | Comment |
|---|---|---|---|---|
|  | OK |  | Preemptive | CallComp is the parent method, declared "capable" of preemptive use; since MyComp is thread-safe internally, CallComp is thread-safe and the process is preemptive |
|  | Error |  | Execution is impossible | CallDial is the parent method, declared "capable"; MyDialog is "indifferent". However, since MyDialog is thread-unsafe internally, it contaminates the call chain. The compilation fails because of a conflict between the declaration of CallDial and its actual capability. The solution is either to modify MyDialog so that it becomes thread-safe so that execution is preemptive, or to change the declaration of CallDial 's property in order to run as cooperative |
|  | OK |  | Cooperative | Since CallDial is declared "incapable" of preemptive use, compilation is thread-unsafe internally; thus execution will always be cooperative, regardless of the status of MyDialog |
|  | OK |  | Cooperative | Since CallComp is the parent method with property "Indifferent", then the process is cooperative even if the entire chain is thread-safe. |
|  | OK |  | Cooperative | Since CallDial is the parent method (property was "Indifferent"), then the process is cooperative and compilation is successful |

### How to find out the actual execution mode

4D allows you to identify the execution mode of processes in compiled mode:

- The **PROCESS PROPERTIES** command allows you to find out whether a process is run in preemptive or cooperative mode.
- Both the Runtime Explorer and the 4D Server administration window display specific icons for preemptive processes (as well as worker processes):

| Process type | Icon |
|---|---|
| Preemptive stored procedure |  |
| Preemptive worker process |  |
| Cooperative worker process |  |

## Writing a thread-safe method

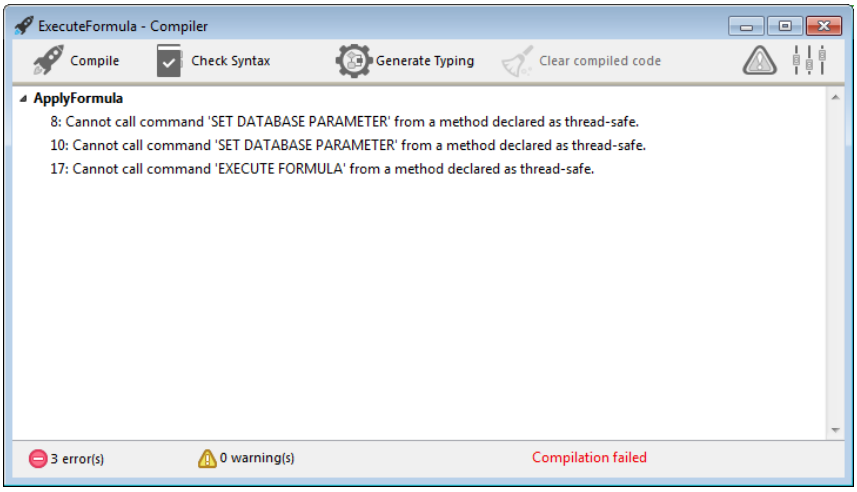
To be thread-safe, a method must respect the following rules:

- It must have either the "Can be run in preemptive processes" or "Indifferent" property
- It must not call a 4D command that is thread-unsafe.
- It must not call another project method that is thread-unsafe
- It must not call a plug-in
- It must not use **Begin SQL/End SQL** code blocks
- It must not use any interprocess variables(*)
- It must not call interface objects(**) (there are exceptions however, see below).

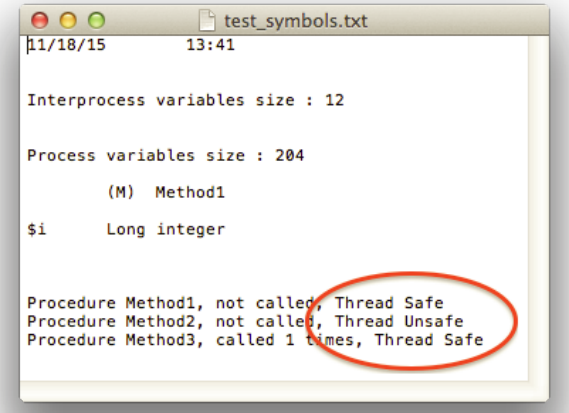
**Note:** In the case of a "Shared by components and host databases" method, the "Can be run in preemptive processes" property must be selected.

(*) Work processes allow you to exchange data between any processes, including preemptive processes. For more information, please refer to the **About workers**.
(**) The **CALL FORM** command provides an elegant solution to call interface objects from a preemptive process.

Methods with the "Can be run in preemptive processes" property will be checked by 4D during compilation. A compilation error is issued whenever the compiler finds something that prevents it from being thread-safe:




The symbol file, if enabled, also contains the thread safety status for each method:




### User interface

Since they are "external" accesses, calls to user interface objects such as forms, as well as to the Debugger, are not allowed in preemptive threads.

The only possible accesses to the user interface from a preemptive thread are:

- Standard error dialog. The dialog is displayed in the user mode process (on 4D single-user) or the server user interface process (4D Server). The **Trace** button is disabled.
- Standard progress indicators
- **ALERT**, **Request** and **CONFIRM** dialogs. The dialog is displayed in the user mode process (on 4D single-user) or the server user interface process (4D Server).

Note that if 4D Server has been launched as a service on Windows with no user interaction allowed, the dialogs will not be displayed.

### Thread-safe 4D commands

A significant number of 4D commands are thread-safe. In the documentation, the  icon in the command property area indicates that the command is thread-safe. You can get a list of thread-safe commands in the *Language Reference* manual. You can also use **Command name** which can return the thread safety property for each command.

### Triggers

When a method uses a command that can call a trigger, the 4D compiler evaluates the thread safety of the trigger in order to check the thread safety of the method:

```
SAVE RECORD([Table_1]) //trigger on Table_1, if it exists, must be thread-safe
```

Here is the list of commands that are checked at compilation time for trigger thread safety:

- SAVE RECORD
- SAVE RELATED ONE
- DELETE RECORD
- DELETE SELECTION
- ARRAY TO SELECTION
- JSON TO SELECTION
- APPLY TO SELECTION
- IMPORT DATA
- IMPORT DIF
- IMPORT ODBC
- IMPORT SYLK
- IMPORT TEXT

If the table is passed dynamically, the compiler may sometimes not be able to find out which trigger it needs to evaluate. Here are some examples of such situations:

```
DEFAULT TABLE([Table_1])
SAVE RECORD
SAVE RECORD($ptrOnTable->)
SAVE RECORD(Table(myMethodThatReturnsATableNumber())->)
```

In this case, all triggers are evaluated. If a thread-unsafe command is detected in at least one trigger, the whole group is rejected and the method is declared thread-unsafe.

### Error-handling methods

Error-catching methods installed by the **ON ERR CALL** command must be thread-safe if they are likely to be called from a preemptive process. In order to handle this case, the compiler now checks the thread safety property of error-catching project methods passed to the **ON ERR CALL** command during compilation and returns appropriate errors if they do not comply with preemptive execution.

Note that this checking is only possible when the method name is passed as a constant, and is not computed, as shown below:

```
ON ERR CALL("myErrMethod1") //will be checked by the compiler
ON ERR CALL("myErrMethod"+String($vNum)) //will not be checked by the compiler
```

In addition, starting with 4D v15 R5, if an error-catching project method cannot be called at runtime (following a thread safety issue, or for any reason like "method not found"), the new error -10532 "Cannot call error handling project method 'methodName'" is generated.

### Pointers compatibility

A process can dereference a pointer to access the value of another process variable only if both processes are cooperative; otherwise, 4D will throw an error. In a preemptive process, if some 4D code tries to dereference a pointer to an interprocess variable, 4D will throw an error.

Example with the following methods:

Method1:

```
myVar:=42
$pid:=New process("Method2";0;"process name";->myVar)
```

Method2:

```
$value:=$1->
```

If either the process running Method1 or the process running Method2 is preemptive, then the expression "$value:=$1->" will throw an execution error.

### DocRef document reference

The use of *DocRef* type parameters (opened document reference, used or returned by **Open document**, **Create document**, **Append document**, **CLOSE DOCUMENT**, **RECEIVE PACKET**, **SEND PACKET**) is limited to the following contexts:

- When called from a preemptive process, a *DocRef* reference is only usable from that preemptive process.
- When called from a cooperative process, a *DocRef* reference is usable from any other cooperative process.

For more information about the *DocRef* reference, please refer to the **DocRef: Document reference number** section.

# Count tasks

Count tasks -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Number of open processes (including kernel processes) |

## Description

**Count tasks** returns the number of processes open in 4D Client, 4D Server (stored procedures) or a single-user version of 4D.

This number takes into account all processes, even those that are automatically managed by 4D. These include the Main process, Design process, Cache Manager process, Indexing process, and Web Server process.

The number returned by **Count tasks** also takes into account processes that have been aborted.

## Example

See the example for **Process state** and for the **On Exit database method**.

# Count user processes

Count user processes -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint |  | Number of live processes (excluding internal processes) |

## Description

**Count user processes** returns the current number of "live" processes in the 4D application whose type is different from -25 (Internal Timer Process), -31 (Client Manager Process) and -15 (Server Interface Process). For more information about process types, please refer to the **PROCESS PROPERTIES** command and to the **Process Type** constants theme.

The **Count user processes** function returns the number of processes opened directly or indirectly by the user (processes for which the *origin* parameter returned by the **PROCESS PROPERTIES** command is greater than or equal to 0).

**Note:** The "live" processes are processes whose status is neither *aborted*, nor *does not exist* (see the **Process state** command).

# Count users

Count users -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint |  | Number of users connected to the server |

## Description

When it is called from a stored procedure on the server, the **Count users** command returns the number of users connected to the server machine.

If the server is running at least one stored procedure and if **Count users** is called from another context (client machine, Web method), the command returns the number of users +1.

In the case of a 4D single-user version, **Count users** returns 1.

# Current process

Current process -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint |  | Process number |

## Description

**Current process** returns the process reference number of the process within which this command is called.

## Example

See the examples for **DELAY PROCESS** and **PROCESS PROPERTIES**.

# Current process name

Current process name -> Function result

| Parameter | Type | Description |
| --- | --- | --- |
| Function result | Text | Name of current process |

## Description

The **Current process name** command returns the name of the process within which this command is called.

This command is particularly useful in the context of worker processes (see the **About workers** section). It can be used to identify the worker process to call when writing generic code.

## Example

You want to call a worker and pass the calling process name as parameter:

```
CALL WORKER(1;"myMessage";Current process name;"Start:"+String(vMax))
```

# DELAY PROCESS

DELAY PROCESS ( process ; duration )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ⇒ | Process number |
| duration | Real | ⇒ | Duration expressed in ticks |

## Description

**DELAY PROCESS** delays the execution of a *process* for a number of ticks (1 tick = 1/60th of a second). During this period, *process* does not take any processing time. Even though the execution of a process may be delayed, it is still in memory.

You can delay a process for less than one tick. For example, if you pass 0.5 in *duration*, the process will be delayed for a 1/2 tick, i.e. 1/120th of a second.

If the process is already delayed, this command delays it again. The *duration* parameter is not added to the time remaining, but replaces it. Therefore pass zero (0) for *duration* if you no longer want to delay a process.

If the process does not exist, the command does nothing.

**Note:** You cannot use this command to assign a stored procedure on the server machine from a client machine (*process*<0).

## Example 1

See example in **Record Locking**.

## Example 2

See example for the command **Process number**.

# EXECUTE ON CLIENT

EXECUTE ON CLIENT ( clientName ; methodName {; param}{; param2 ; ... ; paramN} )

| Parameter | Type | | Description |
|---|---|---|---|
| clientName | String | ➡ | 4D Client's registered name |
| methodName | String | ➡ | Name of the method to execute |
| param | | ➡ | Method's parameter(s) |

## Description

The **EXECUTE ON CLIENT** command forces the execution of the *methodName* method, with the parameters *param1... paramN*, if necessary, on the registered 4D Client whose name is *clientName*. 4D Client's registered name is defined by the **REGISTER CLIENT** command.

This command can be called from a 4D Client or a stored method from 4D Server.

If the method requires one or more parameters, pass them after the name of the method.
The execution of the method on 4D Client is done in a process automatically created on the client workstation, and its name will be the 4D Client's registered name.

If this command is called many times in a row on the same 4D Client, the execution orders will be stacked. Therefore, the methods will be treated one after another in asynchronous mode. The more methods that are stacked, the bigger the workload is for the 4D Client. You can know the state of the workload of each client by using the **GET REGISTERED CLIENTS** command.

**Note:** The stacking of the execution orders cannot be modified or stopped unless 4D Client is unregistered by using the **UNREGISTER CLIENT** command.

You can simultaneously execute the same method on many or all of the registered 4D Clients. To do so, use the wildcard character (@) in the *clientName* parameter.

## Example 1

Let's assume that you want to execute the "GenerateNums" method on the "Client1" client station:

```
EXECUTE ON CLIENT("Client1";"GenerateNums";12;$a;"Text")
```

## Example 2

If you want all the clients to execute the "EmptyTemp" method:

```
EXECUTE ON CLIENT("@";"EmptyTemp")
```

## Example 3

Refer to the example of the **REGISTER CLIENT** command.

## System variables and sets

The **OK** system variable is equal to 1 if 4D Server has correctly received the execution request of a method; however, this does not guarantee that the method has been properly executed by 4D Client.

# Execute on server

Execute on server ( procedure ; stack {; name {; param {; param2 ; ... ; paramN}}}{; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| procedure | String | ➡ | Procedure to be executed within the process |
| stack | Longint | ➡ | Stack size in bytes |
| name | String | ➡ | Name of the process created |
| param | Expression | ➡ | Parameter(s) to the procedure |
| * | Operator | ➡ | Unique process |
| Function result | Longint | ↩ | Process number for newly created process or already executing process |

## Description

The **Execute on server** command starts a new process on the Server machine (if it is called in Client/Server) or on the same machine (if it is called in single-user) and returns the process number for that process.

You use this function to start a stored procedure. For more information about stored procedures, see the section **Stored Procedures** in the 4D Server Reference manual.

If you call **Execute on server** on a Client machine, the command returns a negative process number. If you call it on the Server machine, it returns a positive process number. Note that calling **New process** on the Server machine does the same thing as calling **Execute on server**.

If the process could not be created (for example, if there is not enough memory), **Execute on server** returns zero (0) and an error is generated. You can catch this error using an error-handling method installed using **ON ERR CALL**.

### Process Method

In *procedure*, you pass the name of the process method for the new process. After 4D has set up the context for the new process, it starts executing this method, which therefore becomes the process method.

### Process Stack

The *stack* parameter allows you indicate the amount of memory allocated for the stack of the process. It is the space in memory used to “pile up” method calls, local variables, parameters in subroutines, and stacked records.

- Pass 0 in *stack* to use a default stack size, suitable for most applications (recommended setting).
- In certain particular cases, you may want to use a custom value. It must be expressed in bytes. It is recommended to pass a minimum of 64 KB (around 64,000 bytes) and you case use values above 512 KB in particular if the process can perform large chain calls (subroutines calling subroutines in cascade.

**Note:** The stack is NOT the total memory for the process. Processes share memory for records, interprocess variables, and so on. A process also uses extra memory for storing its process variables. The stack contains various items of 4D internal information; the amount of information kept on the stack depends on the number of nested method calls the process will employ, the number of forms that it will open before closing them and the number and size of local variables used in each nested method call.

**Note for 64-bit 4D Server**: The stack for a process executed on a 64-bit 4D Server usually requires more memory than on a 32-bit 4D Server (about twice as much). Make sure that you check this parameter when your code is intended for execution on a 64-bit 4D Server.

### Process Name

You pass the name of the new process in *name*. In single-user, this name will appear in the list of processes of the Runtime Explorer and will be returned by the **PROCESS PROPERTIES** command when applied to this new process. In Client/Server, this name will appear in blue in the Stored Procedure list of the 4D Server main window.

You can omit this parameter; if you do so, the name of the process will be the empty string.

**Warning:** Contrary to **New process**, do not attempt to make a process local in scope by prefixing its name with the dollar sign ($) while using **Execute on server**. This will work in single-user, because **Execute on server** acts as **New process** in this environment. On the other hand, in Client/Server, this will generate an error.

### Parameter to Process Method

You can pass parameters to the process method. You can pass parameters in the same way as you would pass them to a subroutine. However, there is a restriction—you cannot pass pointer expressions. Also, remember that arrays cannot be

passed as parameters to a method. Upon starting execution in the context of the new process, the process method receives the parameters values in *$1*, *$2*, etc.

**Note:** If you pass parameters to the process method, you must pass the *name* parameter; it cannot be omitted in this case.

If you pass a 4D object (**C_OBJECT**) as param or return value, the JSON form is used in UTF-8 for the server. If the **C_OBJECT** object contains pointers, their dereferenced values are sent, and not the pointers themselves.

### Optional * Parameter

Specifying this last parameter tells 4D to first check whether or not a process with the name you passed in *name* is already running. If it is, 4D does not start a new process and returns the process number of the process with that name.

## Example

The following example shows how importing data can be dramatically accelerated in Client/Server. The ***Regular Import*** method listed below allows you to test how long it takes to import records using the **IMPORT TEXT** command on the Client side:

```
  ` Regular Import Project Method
$vhDocRef:=Open document("")
If(OK=1)
   CLOSE DOCUMENT($vhDocRef)
   FORM SET INPUT([Table1];"Import")
   $vhStartTime:=Current time
   IMPORT TEXT([Table1];Document)
   $vhEndTime:=Current time
   ALERT("It took "+String(0+($vhEndTime-$vhStartTime))+" seconds.")
End if
```

With the regular import data, 4D Client performs the parsing of the text file, then, for each record, create a new record, fills out the fields with the imported data and sends the record to the Server machine so it can be added to the database. There are consequently many requests going over the network. A way to optimize the operation is to use a stored procedure to do the job locally on the Server machine. The Client machine loads the document into a BLOB, start a stored procedure passing the BLOB as parameter. The stored procedure stores the BLOB into a document on the server machine disk, then imports the document locally. The import data is therefore performed locally at a single-user version-like speed because most the network requests have been eliminated. Here is the ***CLIENT IMPORT*** project method. Executed on the Client machine, it starts the *SERVER IMPORT* stored procedure listed just below:

```
  ` CLIENT IMPORT Project Method
  ` CLIENT IMPORT ( Pointer ; String )
  ` CLIENT IMPORT ( -> [Table] ; Input form )

C_POINTER($1)
C_TEXT($2)
C_TIME($vhDocRef)
C_BLOB($vxData)
C_LONGINT(spErrCode)

  ` Select the document do be imported
$vhDocRef:=Open document("")
If(OK=1)
  ` If a document was selected, do not keep it open
   CLOSE DOCUMENT($vhDocRef)
   $vhStartTime:=Current time
  ` Try to load it in memory
   DOCUMENT TO BLOB(Document;$vxData)
   If(OK=1)
  ` If the document could be loaded in the BLOB,
  ` Start the stored procedure that will import the data on the server machine
      $spProcessID:=Execute on server("SERVER IMPORT";0;
      "Server Import Services";Table($1);$2;$vxData)
  ` At this point, we no longer need the BLOB in this process
      CLEAR VARIABLE($vxData)
  ` Wait for the completion of the operation performed by the stored procedure
      Repeat
         DELAY PROCESS(Current process;300)
         GET PROCESS VARIABLE($spProcessID;spErrCode;spErrCode)
         If(Undefined(spErrCode))
```

```
    ` Note: if the stored procedure has not initialized its own instance
    ` of the variable spErrCode, we may be returned an undefined variable
              spErrCode:=1
           End if
        Until(spErrCode<=0)
    ` Tell the stored procedure that we acknowledge
        spErrCode:=1
        SET PROCESS VARIABLE($spProcessID;spErrCode;spErrCode)
        $vhEndTime:=Current time
        ALERT("It took "+String(0+($vhEndTime-$vhStartTime))+" seconds.")
     Else
        ALERT("There is not enough memory to load the document.")
     End if
  End if
```

Here is the ***SERVER IMPORT*** project method executed as a stored procedure:

```
    ` SERVER IMPORT Project Method
    ` SERVER IMPORT ( Long ; String ; BLOB )
    ` SERVER IMPORT ( Table Number ; Input form ; Import Data )

  C_LONGINT($1)
  C_TEXT($2)
  C_BLOB($3)
  C_LONGINT(spErrCode)

    ` Operation is not finished yet, set spErrCode to 1
  spErrCode:=1
  $vpTable:=Table($1)
  FORM SET INPUT($vpTable->;$2)
  $vsDocName:="Import File "+String(1+Random)
  DELETE DOCUMENT($vsDocName)
  BLOB TO DOCUMENT($vsDocName;$3)
  IMPORT TEXT($vpTable->;$vsDocName)
  DELETE DOCUMENT($vsDocName)
    ` Operation is finished, set spErrCode to 0
  spErrCode:=0
    ` Wait until the requester Client got the result back
  Repeat
     DELAY PROCESS(Current process;1)
  Until(spErrCode>0)
```

Once these two project methods have been implemented in a database, you can perform a “Stored Procedure-based” import data by, for instance, writing:

```
  CLIENT IMPORT(->[Table1];"Import")
```

With some benchmarks you will discover that using this method you can import records up to 60 times faster than the regular import.

# GET REGISTERED CLIENTS

GET REGISTERED CLIENTS ( clientList ; methods )

| Parameter | Type | | Description |
|---|---|---|---|
| clientList | Text array | ⇦ | List of the saved 4D Clients |
| methods | Longint array | ⇦ | List of the methods to be executed |

## Description

The **GET REGISTERED CLIENTS** command fills two arrays:

- *clientLists* contains the list of clients who were “registered” by using the **REGISTER CLIENT** command.
- *methods* supplies the list of each client’s “workload”. The workload is the number of methods that a 4D Client must still execute by calling the **EXECUTE ON CLIENT** command (for more information, please refer to the description of the **EXECUTE ON CLIENT** command).

## Example 1

Let’s assume that you want to obtain a list of all the registered clients and the methods that remain to be executed:

```
ARRAY TEXT($clients;0)
ARRAY LONGINT($methods;0)
GET REGISTERED CLIENTS($clients;$methods)
```

## Example 2

Refer to the example of the **REGISTER CLIENT** command.

## System variables and sets

If the operation was successful, the *OK* system variable is equal to 1.

# New process

New process ( method ; stack {; name {; param {; param2 ; ... ; paramN}}}{; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| method | String | ➡ | Method to be executed within the process |
| stack | Longint | ➡ | Stack size in bytes |
| name | String | ➡ | Name of the process created |
| param | Expression | ➡ | Parameter(s) to the method |
| * | Operator | ➡ | Unique process |
| Function result | Longint | ↩ | Process number for newly created process or already executing process |

## Description

The **New process** command starts a new process (on the same machine) and returns the process number for that process.

If the process could not be created (for example, if there is not enough memory), **New process** returns zero (0) and an error is generated. You can catch this error using an error-handling method installed using **ON ERR CALL**.

### Process Method

In *method*, you pass the name of the process method for the new process. After 4D has set up the context for the new process, it starts executing this method, which therefore becomes the process method.

### Process Stack

The *stack* parameter allows you to indicate the amount of memory allocated for the stack of the process. This is the space in memory used to "pile up" method calls, local variables, parameters in subroutines, and stacked records.

- Pass 0 in *stack* to use a default stack size, suitable for most applications (recommended setting).
- In certain particular cases, you may want to use a custom value. It must be expressed in bytes. It is recommended to pass a minimum of 64 KB (around 64,000 bytes) and you case use values above 512 KB in particular if the process can perform large chain calls (subroutines calling subroutines in cascade.

**Note:** The stack is NOT the total memory for the process. Processes share memory for records, interprocess variables, and so on. A process also uses extra memory for storing its process variables. The stack contains various items of 4D information; the amount of information kept on the stack depends on the number of nested method calls the process will employ, the number of forms that it will open before closing them and the number and size of local variables used in each nested method call.

**Note for 64-bit 4D Server**: The stack for a process executed on a 64-bit 4D Server usually requires more memory than on a 32-bit 4D Server (about twice as much). Make sure that you check this parameter when your code is intended for execution on a 64-bit 4D Server.

### Process Name

You pass the name of the new process in *name*. This name will appear in the list of processes of the Runtime Explorer and will be returned by the **PROCESS PROPERTIES** command when applied to this new process. You can omit this parameter; if you do so, the name of the process will be the empty string. You can make a process local in scope by prefixing its name with the dollar sign (*$*).

**Important:** Remember that local processes should not access data in Client/Server.

### Parameters to Process Method

You can pass parameters to the process method using one or more *param* parameters. You pass parameters in the same way as you would pass them to a subroutine (see the **Passing Parameters to Methods** section). Upon starting execution in the context of the new process, the process method receives the parameters values in *$1*, *$2*, etc. Remember that arrays cannot be passed as parameters to a method. Furthermore, these additional considerations are to be taken into account in the context of the **New process** command:

- pointers to tables or fields are allowed.
- pointers to variables, particularly local and process variables, are not recommended since these variables may be undefined at the time when they are being accessed by the process method.
- if you pass an Object type parameter, 4D will create in this case a copy of the object in the destination process.

**Note:** If you pass parameters to the process method, you must pass the *name* parameter; it cannot be omitted in this case.

### Optional * Parameter

Specifying this last parameter tells 4D to first check whether or not a process with the name you passed in *name* is already running. If it is, 4D does not start a new process and returns the process number of the process with that name.

## Example

Given the following project method:

```
  ` ADD CUSTOMERS
SET MENU BAR(1)
Repeat
   ADD RECORD([Customers];*)
Until(OK=0)
```

If you attach this project method to a custom menu item **Menu Bar Editor** window whose **Start a New Process** property is set, 4D will automatically start a new process running that method. The call **SET MENU BAR**(1) adds a menu bar to the new process. In the absence of any window (that you could open with **Open window**), the call to **ADD RECORD** will automatically open one.

To be able to start this Add Customers process when you click on a button in a custom control panel, you can write:

```
  ` bAddCustomers button object method
$vlProcessID:=New process("Add Customers";0;"Adding Customers")
```

The button does the same thing as the custom menu item.

While choosing the menu item or clicking the button, if you want to start the process (if it does not exist) or bring it to the front (if it is already running), you can create the method ***START ADD CUSTOMERS***:

```
  ` START ADD CUSTOMERS
$vlProcessID:=New process("Add Customers";0;"Adding Customers";*)
If($vlProcessID#0)
   BRING TO FRONT($vlProcessID)
End if
```

The object method of the *bAddCustomers* becomes:

```
  ` bAddCustomers button object method
START ADD CUSTOMERS
```

In the Menu Bar editor, you replace the method ***ADD CUSTOMERS*** with the method ***START ADD CUSTOMERS***, and you deselect the **Start a New Process** property for the menu item.

# PAUSE PROCESS

PAUSE PROCESS ( process )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ⇨ | Process number |

## Description

**PAUSE PROCESS** suspends the execution of *process* until it is reactivated by the **RESUME PROCESS** command. During this period, *process* does not take any time on your machine. Even though a process may be paused, the process is still in memory.

If *process* is already paused, **PAUSE PROCESS** does nothing. If the process has been delayed using the **DELAY PROCESS** command, the process is paused. **RESUME PROCESS** resumes the process immediately.

While process execution is suspended, the windows belonging to this process are not enterable. In this case, to avoid confusing the user, consider hiding the process. If *process* does not exist, the command does nothing.

**Warning:** Use **PAUSE PROCESS** only in processes that you have started. It will not affect the main process.

**Note:** You cannot use this command to assign a stored procedure on the server machine from a client machine (*process*<0).

# Process aborted

Process aborted -> Function result

| Parameter | Type | Description |
| --- | --- | --- |
| Function result | Boolean | True = the process is about to be aborted, False = the process is not about to be aborted |

## Description

The **Process aborted** command returns **True** if the process in which it is called is about to be interrupted unexpectedly, which means that the execution of the command was unable to reach its “normal” completion. For example, this can occur after calling **QUIT 4D**.

# Process number

Process number ( name {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| name | String | ⇒ | Name of process for which to retrieve the process number |
| * | | ⇒ | Return the process number from the server |
| Function result | Longint | ⇐ | Process number |

## Description

**Process number** returns the number of the process whose name you pass in *name*. If no process is found, Process number returns 0.

The optional parameter * allows you to retrieve, from 4D Client, the process ID of a process that is executed on the server (a stored procedure). In this case, the returned value is negative. This option is especially useful when using the **GET PROCESS VARIABLE**, **SET PROCESS VARIABLE** and **VARIABLE TO VARIABLE** commands. Please refer to the descriptions of these commands for details.

If the command is executed with the * parameter from a process on the server machine, the returned value is positive.

## Example

You create a custom floating window, run in a separate process, in which you implement your own tools to interact with the Design environment. For example, when selecting an item in a hierarchical list of keywords, you want to paste some text into the frontmost window of the Design environment. To do so, you can use the pasteboard, but the pasting event must occur within the Design process. The following small function returns the process number of the Design process (if running):

```
  ` Design process number Project Method
  ` Design process number -> LongInt
  ` Design process number -> Design process number

$0:=Process number("Design Process")
  ` Note: This can break in the future if the process name changes
```

Using this function, the following project method pastes the text received as parameter to the frontmost window of the Design environment (if applicable):

```
  ` PASTE TEXT TO DESIGN Project Method
  ` PASTE TEXT TO DESIGN ( Text )
  ` PASTE TEXT TO DESIGN ( Text to Paste in frontmost Design window )

C_TEXT($1)
C_LONGINT($vlDesignPID;$vlCount)

$vlDesignPID:=Design process number
If($vlDesignPID #0)
  ` Put the text into the pasteboard
   SET TEXT TO PASTEBOARD($1)
  ` Post a Ctrl-V / Command-V event
   POST KEY(Character code("v");Command key mask;$vlDesignPID)
  ` Call repeatedly DELAY PROCESS so the scheduler gets a chance
  ` to pass over the event to the Design process
   For($vlCount;1;5)
      DELAY PROCESS(Current process;1)
   End for
End if
```

# PROCESS PROPERTIES

PROCESS PROPERTIES ( process ; procName ; procState ; procTime {; procMode {; uniqueID {; origin}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ⇒ | Process number |
| procName | String | ⇐ | Process name |
| procState | Longint | ⇐ | Process state |
| procTime | Longint | ⇐ | Cumulative time taken by process in ticks |
| procMode | Boolean, Longint | ⇐ | If Boolean: Visible (True) or Hidden (False)<br>If Longint (bit field): bit 0 = Visibility, bit 1 = Preemptive execution |
| uniqueID | Integer | ⇐ | Unique process number |
| origin | Longint | ⇐ | Origin of the process |

## Description

The **PROCESS PROPERTIES** command returns various information about the process whose process number you pass in *process*.

After the call:

- *procName* returns the name of the process. Some things to note about the process name:
  - If the process was started from the **Execute Method** dialog box (with the **New Process** option selected), its name is “**P_**” followed by a number.
  - If the process was started from a custom menu item whose **Start a New Process** property is checked, the name of the process is “M_” or “ML_” followed by a number.
  - If the process was started by the Web server, its name is "Web Process#" followed by an UUID.
  - If the process has been aborted (and its “slot” not reused yet), the name of the process is still returned. To detect if a process is aborted, test *procState=-1* (see below).
- *procState* returns the state of the process at the moment of the call. This parameter can return one of the values provided by the following predefined constants:

| Constant | Type | Value |
|---|---|---|
| Aborted | Longint | -1 |
| Delayed | Longint | 1 |
| Does not exist | Longint | -100 |
| Executing | Longint | 0 |
| Hidden modal dialog | Longint | 6 |
| Paused | Longint | 5 |
| Waiting for input output | Longint | 3 |
| Waiting for internal flag | Longint | 4 |
| Waiting for user event | Longint | 2 |

- *procTime* returns the cumulative time that the process has used since it started, in ticks (1/60th of a second) .
- The optional *procMode* parameter can be a Boolean or longint type variable:
  - If it is a Boolean type, it returns True if the process is visible and False if it is hidden.
  - If it is a longint type, after the method is executed, it contains a bit field where the two first bits are set:
    - bit 0 returns the visibility property: set to 1 if process is visible, and 0 if it is hidden
    - bit 1 returns the preemptive mode property: set to 1 if process runs in preemptive mode, and 0 if it runs in cooperative mode.

      **Note:** This property is only useful in 64-bit 4D applications, where processes can be run preemptively or cooperatively. For more information, please refer to the **Preemptive 4D processes** section.
- *uniqueID*, if specified, returns the unique process number. Actually, each process has attributed a process number to it as well as a unique process number per session. The unique number allows you to differentiate between two processes or two process sessions. It corresponds to the process number having been started during 4D’s session.
- *origin*, if specified, returns a value that describes the origin of the process. 4D offers the following predefined constants (in the "**Process Type**" theme):

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_Web process with context | Longint | -11 | |
| Apple event manager | Longint | -7 | |
| Backup process | Longint | -19 | |
| Cache manager | Longint | -4 | |
| Client manager process | Longint | -31 | |
| Created from execution dialog | Longint | 3 | |
| Created from menu command | Longint | 2 | |
| Design process | Longint | -2 | |
| Event manager | Longint | -8 | |
| Execute on client process | Longint | -14 | |
| Execute on server process | Longint | 1 | |
| External task | Longint | -9 | |
| Indexing process | Longint | -5 | |
| Internal 4D server process | Longint | -18 | |
| Internal timer process | Longint | -25 | |
| Log file process | Longint | -20 | |
| Main process | Longint | -1 | |
| Method editor macro process | Longint | -17 | |
| Monitor process | Longint | -26 | |
| MSC process | Longint | -22 | |
| None | Longint | 0 | |
| On exit process | Longint | -16 | |
| Other 4D process | Longint | -10 | |
| Other user process | Longint | 4 | |
| Restore Process | Longint | -21 | |
| Serial Port Manager | Longint | -6 | |
| Server interface process | Longint | -15 | |
| SQL Method execution process | Longint | -24 | |
| Web process on 4D remote | Longint | -12 | |
| Web process with no context | Longint | -3 | |
| Web server process | Longint | -13 | |
| Worker process | Longint | 5 | Worker process launched by user |

**Note:** 4D's internal processes return a negative value and the processes generated by the user return a positive value.

If the process does not exist, which means you did not pass a number in the range 1 to **Count tasks**, **PROCESS PROPERTIES** leaves the variable parameters unchanged.

## Example 1

The following example returns the name, state, and time taken in the variables *vName*, *vState,* and *vTimeSpent* for the current process:

```
C_STRING(80;vName) ` Initialize the variables
C_INTEGER(vState)
C_INTEGER(vTime)
PROCESS PROPERTIES(Current process;vName;vState;vTimeSpent)
```

## Example 2

See example for **On Exit database method**.

## Example 3

You want to find out the visibility and execution mode of the current process. You can write:

```
C_TEXT(vName)
C_LONGINT(vState)
C_LONGINT(vTime)
C_LONGINT(vFlags)
C_BOOLEAN(isVisible)
C_BOOLEAN(isPreemptive)
PROCESS PROPERTIES(Current process;vName;vState;vTime;vFlags)
isVisible:=vFlags?? 0 //true if visible
isPreemptive:=vFlags?? 1 //true if preemptive
```

# Process state

Process state ( process ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ➡ | Process number |
| Function result | Longint | ⮌ | State of the process |

## Description

The **Process state** command returns the state of the process whose number you pass in *process*.

The function result can be one of the values provided by the following predefined constants:

| Constant | Type | Value |
|---|---|---|
| Aborted | Longint | -1 |
| Delayed | Longint | 1 |
| Does not exist | Longint | -100 |
| Executing | Longint | 0 |
| Hidden modal dialog | Longint | 6 |
| Paused | Longint | 5 |
| Waiting for input output | Longint | 3 |
| Waiting for internal flag | Longint | 4 |
| Waiting for user event | Longint | 2 |

If the process does not exist (which means you did not pass a number in the range 1 to **Count tasks**), **Process state** returns Does not exist (-100).

## Example

The following example puts the name and process reference number for each process into the *asProcName* and *aiProcNum* arrays. The method checks to see if the process has been aborted. In this case, the process name and number are not added to the arrays:

```
$vlNbTasks:=Count tasks
ARRAY TEXT(asProcName;$vlNbTasks)
ARRAY INTEGER(aiProcNum;$vlNbTasks)
$vlActualCount:=0
For($vlProcess;1;$vlNbTasks)
   If(Process state($vlProcess)>=Executing)
      $vlActualCount:=$vlActualCount+1
      PROCESS PROPERTIES($vlProcess;asProcName{$vlActualCount};$vlState;$vlTime)
      aiProcNum{$vlActualCount}:=$vlProcess
   End if
End for
 ` Eliminate unused extra elements
ARRAY TEXT(asProcName;$vlNbTasks)
ARRAY INTEGER(aiProcNum;$vlActualCount)
```

# REGISTER CLIENT

REGISTER CLIENT ( clientName {; period}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| clientName | String | ➡ | Name of the 4D client session |
| period | Longint | ➡ | ***Ignored since version 11.3*** |
| * | Operator | ➡ | Local process |

## Description

The **REGISTER CLIENT** command “registers” a 4D client station with the name specified in *clientName* on 4D Server, so as to allow other clients or eventually 4D Server (by using stored methods) to execute methods on it by using the **EXECUTE ON CLIENT** command. Once it is registered, a 4D client can then execute one or more methods for other clients.

**Notes:**

- You can also automatically register each client station that connects to 4D Server by using the “Register Clients at Startup...” option in the Preferences dialog box.
- If this command is used with 4D in local mode, it has no effect.
- More than one 4D client station can have the same registered name.

When this command is executed, a process, named *clientName*, is created on the client station. This process can only be aborted by the **UNREGISTER CLIENT** command.
If you pass the optional * parameter, the created process is local. 4D will automatically add the dollar sign ($) at the beginning of the process name. Otherwise, the process is global.

**Compatibility Note:** Since version 11.3 of 4D, the server/client communication mechanisms have been optimized. Now the server sends execution requests directly to the registered clients when necessary (technology "push"). The previous principle where clients queried the server periodically is no longer used. The *period* parameter is ignored if it is passed.

Once the command is executed, it is not possible to modify a 4D client’s name on the fly. To do so, you must call the **UNREGISTER CLIENT** command, then the **REGISTER CLIENT** command.

## Example

In the following example, we are going to create a small messaging system that allows the client workstations to communicate between themselves.

1) This method, Registration, allows you to register a 4D client and to keep it ready to receive a message from another 4D client:

```
  `You must unregister before registering under another name
UNREGISTER CLIENT
Repeat
   vPseudoName:=Request("Enter your name:";"User";"OK";"Cancel")
Until((OK=0)|(vPseudoName#""))
If(OK=0)
   ... ` Don't do anything
Else
   REGISTER CLIENT(vPseudoName)
End if
```

2) The following instruction allows you to get a list of the registered clients. It can be placed in the **On Startup database method**:

```
PrClientList:=New process("4D Client List";32000;"List of registered clients")
```

3) The method 4D Client List allows you to recuperate all the registered 4D clients and those that can receive messages:

```
If(Application type=4D Remote Mode)
  ` the code below is only valid in client-server mode
   $Ref:=Open window(100;100;300;400;-(Palette window+Has window title);"List of registered clients")
   Repeat
```

```
      GET REGISTERED CLIENTS($ClientList;$ListeCharge)
   `Retrieve the registered clients in $ClientList
      ERASE WINDOW($Ref)
      GOTO XY(0;0)
      For($p;1;Size of array($ClientList))
         MESSAGE($ClientList{$p}+Char(Carriage return))
      End for
   `Display each second
      DELAY PROCESS(Current process;60)
   Until(False) ` Infinite loop
End if
```

4) The following method allows you to send a message to another registered 4D client. It calls the Display_Message method (see below).

```
$Addressee:=Request("Addressee of the message:";"")
 ` Enter the name of the people visible in the window generated by the
 ` On Startup database method
If(OK#0)
   $Message:=Request("Message:") ` message
   If(OK#0)
      EXECUTE ON CLIENT($Addressee;"Display_Message";$Message) ` Send message
   End if
End if
```

5) Here is the Display_Message method:

```
C_TEXT($1)
ALERT($1)
```

6) Finally, this method allows a client station to no longer be visible by the other 4D clients and to no longer receive messages:

```
UNREGISTER CLIENT
```

## System variables and sets

If the 4D client is correctly registered, the **OK** system variable is equal to 1. If the 4D client was already registered, the command doesn't do anything and **OK** is equal to 0.

# RESUME PROCESS

RESUME PROCESS ( process )

| Parameter | Type | | Description |
|---|---|---|---|
| process | Longint | ➡ | Process number |

## Description

**RESUME PROCESS** resumes a *process* whose execution has been paused or delayed. If *process* is not paused or delayed, the command does nothing.

If *process* has been delayed before, see the **PAUSE PROCESS** or **DELAY PROCESS** commands. If *process* does not exist, the command does nothing.

**Note:** You cannot use this command to assign a stored procedure on the server machine from a client machine (*process*<0).

# UNREGISTER CLIENT

UNREGISTER CLIENT
Does not require any parameters

## Description

The **UNREGISTER CLIENT** command “unregisters” a 4D client station. The client must have already been registered by the **REGISTER CLIENT** command.

**Note:** A 4D client is automatically unregistered when the user quits the application.

If the client workstation was not previously registered or if the command was executed on 4D in local mode, the command has no effect.

## Example

Refer to the example for the **REGISTER CLIENT** command.

## System variables and sets

If the client is correctly unregistered, the OK system variable is set to 1. If the client was not registered, OK is set to 0.

# Queries

- DESCRIBE QUERY EXECUTION
- Find in field
- Get last query path
- Get last query plan
- GET QUERY DESTINATION
- Get query limit
- ORDER BY
- ORDER BY FORMULA
- QUERY
- QUERY BY ATTRIBUTE
- QUERY BY EXAMPLE
- QUERY BY FORMULA
- QUERY SELECTION
- QUERY SELECTION BY ATTRIBUTE New 16.0
- QUERY SELECTION BY FORMULA
- QUERY SELECTION WITH ARRAY
- QUERY WITH ARRAY
- SET QUERY AND LOCK
- SET QUERY DESTINATION
- SET QUERY LIMIT

# DESCRIBE QUERY EXECUTION

DESCRIBE QUERY EXECUTION ( status )

| Parameter | Type | | Description |
|---|---|---|---|
| status | Boolean | ⇒ | True=Enable internal query analysis, False=Disable internal query analysis |

## Description

The **DESCRIBE QUERY EXECUTION** command enables or disables the query analysis mode for the current process. The command only works in the context of 4D language query commands such as **QUERY**.

Calling the command with the *status* parameter set to **True** enables the query analysis mode. In this mode, the 4D engine records internally two specific pieces of information for each subsequent query carried out on the data:

- A detailed internal description of the query just before its execution, in other words, what was planned to be executed (the query plan),
- A detailed internal description of the query that was actually executed (the query path).

The information recorded includes the type of query (indexed, sequential), the number of records found and the time needed for every query criteria to be executed. Y ou can then read this information using the **Get last query plan** and **Get last query path** commands.

Usually, the description of the query plan and its path are the same, but they may nevertheless differ because 4D might implement dynamic optimizations during the query execution in order to improve performance. For example, an indexed query may be converted dynamically into a sequential query if the 4D engine estimates that this might be faster — this is sometimes the case, more particularly, when the number of records being queries is low.

Pass **False** in the *status* parameter when you no longer need to analyze queries. The query analysis mode can slow down the application.

## Example

The following example illustrates the type of information obtained using these commands:

```
C_TEXT($vResultPlan;$vResultPath)
DESCRIBE QUERY EXECUTION(True) //analysis mode
QUERY([Employees];[Employees]LastName="T@";*) // Search for employees whose last name starts with T...
QUERY([Employees]; & ;[Companies]Name="H@";*) // that work for a company whose name starts with H
QUERY([Employees]; & ;[Employees]Salary>2500;*) // whose salary is > 2500
QUERY([Employees]; & ;[Cities]Pop<50000) // that live in a city with less than 50,000 inhabitants
$vResultPlan:=Get last query plan(Description in text format)
$vResultPath:=Get last query path(Description in text format)
DESCRIBE QUERY EXECUTION(False) //End of analysis mode
```

After executing this code, *$vResultPlan* and *$vResultPath* contain descriptions of the queries performed, for example:

```
$vResultPlan :
   Employees.LastName == T@ And Employees.Salary > 2500 And Join on Table : Companies  :  Employees.Company =
Companies.Name [index : Companies.Name ] LIKE H@ And Join on Table : Cities  :  Employees.City = Cities.Name [index :
Cities.Pop ] < 50000
$vResultPath :
(Employees.LastName == T@ And Employees.Salary  > 2500) And (Join on Table : Companies  :  Employees.Company  =
Companies.Name with filter {[index : Companies.Name ] LIKE H@}) And (Join on Table : Cities  :  Employees.City =
Cities.Name with filter {[index : Cities.Pop ] < 50000})   (3 records found in 1 ms)
```

If the Description in XML Format constant is passed to the **Get last query path** command, *$vResultPath* contains the description of the query expressed in XML:

```
$vResultPath :   <QueryExecution>       <steps description="And" time="0" recordsfounds="1227">          <steps
description="[Merge] : ACTORS with CITIES" time="13" recordsfounds="1227">            <steps description="[Join] :
ACTORS.Birth_City_ID =CITIES.City_ID" time="13" recordsfounds="1227"/>          </steps>       </steps>
   </QueryExecution>
```

# Find in field

Find in field ( targetField ; value ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| targetField | Field | ⇒ | Field on which to execute the search |
| value | Field, Variable | ⇒ | Value to search |
| | | ⇐ | Value found |
| Function result | Longint | ⇒ | Number of the record found or -1 if no record was found |

## Description

The **Find in field** command returns the number of the first record whose *targetField* field is equal to *value*.
If no records are found, **Find in field** returns -1.

After calling this command, *value* contains the value found. This feature allows you to execute searches using the wildcard character (“@”) on Alpha fields and then retrieve the value found.

**Note:** Due to this principle, you cannot use a *parameter* ($1, $2, etc.) in *value* because this would cause malfunctions in compiled mode. Similarly, if you pass a field in the *value* parameter, keep in mind that its value will be reassigned if the query is successful (the command **Modified record**, in particular, will return True for the current record of the table).

This command doesn’t modify the current selection or the current record.
It is fast and particularly useful to avoid creating double entries during data entry.

**Historical note:** In earlier versions of 4D, the **Find in field** command was named **Find index key** and only worked with indexed fields. Beginning with 4D v11 SQL, this limitation was removed and the command was renamed.

## Example 1

In an audio CD database, during data entry let’s assume that you want to verify the singer’s name to see if it already exists in the database. Because homonyms can exist, you don’t want the [Singer]Name field to be unique. Therefore, in the input form, you can write the following code in the [Singer]Name field’s object method:

```
If(Form event=On Data Change)
   $RecNum:=Find in field([Singer]Name;[Singer]Name)
   If($RecNum #-1) ` If this name has already been entered
      CONFIRM("A singer with the same already exists. Do you want to see the record?";"Yes";"No")
      If(OK=1)
         GOTO RECORD([Singer];$RecNum)
      End if
   End if
End if
```

## Example 2

Here is an example that lets you verify the existence of a value:

```
C_LONGINT($id;$1)
$id:=$1
If(Find in field([MyTable]MyID;$id)>=0)
   $0:=True
Else
   $0:=False
End if
```

Note the >= that lets you cover all cases. In fact, the function returns a record number and the first record is numbered 0.

# Get last query path

Get last query path ( descFormat ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| descFormat | Longint | ⇒ | Description format (Text or XML) |
| Function result | String | ⇐ | Description of last executed query path |

## Description

The **Get last query path** command returns the detailed internal description of the actual path of the last query carried out on the data. For more information about query descriptions, please refer to the documentation of the **DESCRIBE QUERY EXECUTION** command.

This description is returned in Text or XML format depending on the value passed in the *descFormat* parameter. You can pass one of the following constants, found in the "**Queries**" theme:

| Constant | Type | Value |
|---|---|---|
| Description in text format | Longint | 0 |
| Description in XML format | Longint | 1 |

This command returns a significant value if the **DESCRIBE QUERY EXECUTION** command has been executed during the session.

The description of the last query path can be compared to the description of the query plan provided for the last query (obtained using the **Get last query plan** command) for optimization purposes.

# Get last query plan

Get last query plan ( descFormat ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| descFormat | Longint | ⇒ | Description format (Text or XML) |
| Function result | String | ⇐ | Description of last executed query plan |

## Description

The **Get last query plan** command returns the detailed internal description of the query plan for the last query carried out on the data. For more information about query descriptions, please refer to the documentation of the **DESCRIBE QUERY EXECUTION** command.

This description is returned in Text or XML format depending on the value passed in the *descFormat* parameter. You can pass one of the following constants, found in the "**Queries**" theme:

| Constant | Type | Value |
|---|---|---|
| Description in text format | Longint | 0 |
| Description in XML format | Longint | 1 |

This command returns a significant value if the **DESCRIBE QUERY EXECUTION** command has been executed during the session.

The description of the last query plan can be compared to the description of the actual path of the last query (obtained using the **Get last query path** command) for optimization purposes.

# GET QUERY DESTINATION

GET QUERY DESTINATION ( destinationType ; destinationObject ; destinationPtr )

| Parameter | Type | | Description |
|---|---|---|---|
| destinationType | Longint | ⇦ | 0=current selection, 1=set, 2=named selection, 3=variable |
| destinationObject | String | ⇦ | Name of the set or Name of the named selection or Empty string |
| destinationPtr | Pointer | ⇦ | Pointer to local variable if destinationType=3 |

## Description

The **GET QUERY DESTINATION** command returns the current destination of query results for the process underway. By default, query results modify the current selection, but you can change this using the **SET QUERY DESTINATION** command.

In the *destinationType* parameter, 4D returns a value indicating the current destination of queries and in the *destinationObject* parameter it returns the name of the destination (if applicable). You can compare the value of the *destinationType* parameter with the constants of the **Queries** theme:

| Constant | Type | Value |
|---|---|---|
| Into current selection | Longint | 0 |
| Into named selection | Longint | 2 |
| Into set | Longint | 1 |
| Into variable | Longint | 3 |

The value returned in the *destinationObject* parameter depends on the value of the *destinationType* parameter:

| destinationType parameter | destinationObject parameter |
|---|---|
| 0 (current selection ) | *destinationObject* is an empty string |
| 1 (set) | *destinationObject* contains the name of the set |
| 2 (named selection ) | *destinationObject* contains the name of the selection |
| 3 (variable) | *destinationObject* is an empty string (use the *destinationPtr* parameter) |

When the query destination is a local variable (*destinationType* returns 3), 4D returns a pointer to this variable in the *destinationPtr* parameter.

## Example

We want to modify the query destination temporarily and then restore the previous parameters:

```
GET QUERY DESTINATION($vType;$vName;$ptr)
 //retrieval of current parameters
SET QUERY DESTINATION(Into set;"$temp")
 //temporary modification of destination
QUERY(...) //query
SET QUERY DESTINATION($vType;$vName;$ptr)
 //restoring parameters
```

# Get query limit

Get query limit -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint |  | Limit number of records,<br>0 = unlimited number |

## Description

The **Get query limit** command returns the limit for the number of records that a query may find in the current process.

You set this limit using the **SET QUERY LIMIT** command.

By default, if no limit is set, the command returns 0.

# ORDER BY

ORDER BY ( {aTable ;}{ aField }{; > or < }{; aField2 ; > or <2 ; ... ; aFieldN ; > or <N}{; *} )

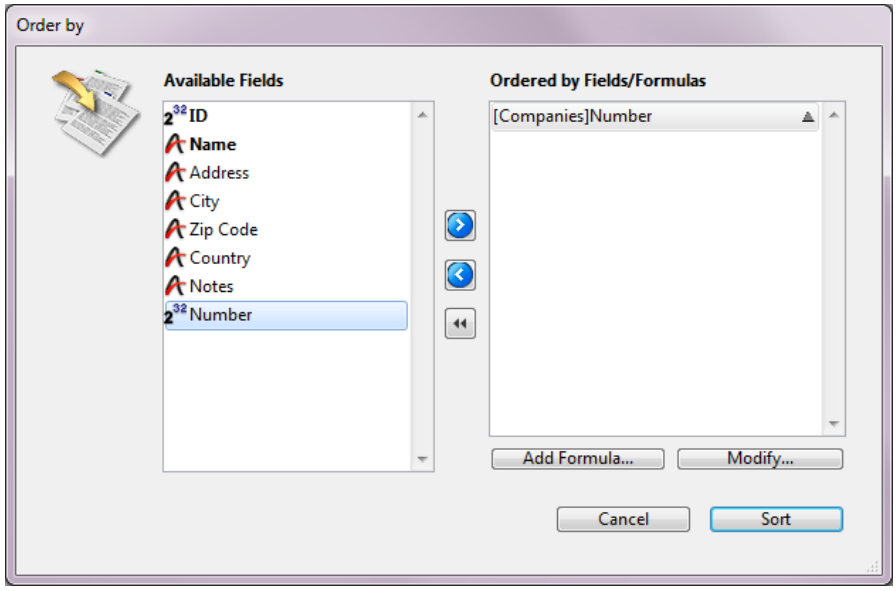
| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to order selected records, or Default table, if omitted |
| aField | Field | ⇒ | Field on which to set the order for each level |
| > or < | Operator | ⇒ | Ordering direction for each level: > to order in ascending order, or < to order in descending order |
| * | Operator | ⇒ | Continue order flag |

## Description

**ORDER BY** sorts (reorders) the records of the current selection of *aTable* for the current process. After the sort has been completed, the new first record of the selection becomes the current record.

If you omit the *aTable* parameter, the command applies to the default table, if it has been specified. Otherwise, 4D uses the table of the first field passed as a parameter. If you do not pass a parameter and if no default table has been specified, an error is returned.

If you do not specify the *aField*, the *> or <* or the * parameters, **ORDER BY** displays the Order By editor for *aTable*. The Order By editor is shown here:




For more information about using the Order By editor, refer to the 4D Design Reference manual.

If you specify the *aField* and *> or <* parameters, the standard Order By editor is not presented and the sort is defined programmatically. You can sort the selection on one level or on several levels. For each sort level, you specify a field in *aField* and the sorting order in *> or <*. If you pass the “greater than” symbol (>), the order is ascending. If you pass the “less than” symbol (<), the order is descending.
If you omit the sorting order parameter *> or <*, ascending order is the default.

If only one field is specified (one level sort) and it is indexed, the index is used for the order. If the field is not indexed or if there is more than one field, the order is performed sequentially (except in the case of composite indexes). The field may belong to the (selection’s) table being reordered or to a One table related to *aTable* with an automatic or manual relation. In this case, the sort is always sequential.
If the sorted fields are included in a composite index, **ORDER BY** uses the index for the order.

For multiple sorts (sorts on multiple fields), you can call **ORDER BY** as many times as necessary and specify the optional * parameter, except for the last **ORDER BY** call, which starts the actual sort operation. This feature is useful for multiple sorts management in customized user interfaces.
**Warning:** with this syntax, you can pass only one sort level (field) per **ORDER BY** call.

No matter what way a sort has been defined, if the actual sort operation is going to take some time to be performed, 4D automatically displays a message containing a progress thermometer. These messages can be turned on and off by using the commands **MESSAGES ON** and **MESSAGES OFF**. If the progress thermometer is displayed, the user can click the Stop button to interrupt the sort.

If the sort is performed without interruption, the OK variable is set to *1*. If the user clicks Cancel or Stop, the **ORDER BY** terminates with no sort actually performed, and sets the OK variable to *0* (zero).

**Note:** This command does not support Object type fields.

## Example 1

The following line displays the Order By editor for the [Products] table:

```
ORDER BY([Products])
```

## Example 2

The following line displays the Order By editor for the default table (if it has been set):

```
ORDER BY
```

## Example 3

The following line orders the selection of [Products] by name in ascending order:

```
ORDER BY([Products];[Products]Name;>)
```

## Example 4

The following line orders the selection of [Products] by name in descending order:

```
ORDER BY([Products];[Products]Name;<)
```

## Example 5

The following line orders the selection of [Products] by type and price in ascending order for both levels:

```
ORDER BY([Products];[Products]Type;>;[Products]Price;>)
```

## Example 6

The following line orders the selection of [Products] by type and price in descending order for both levels:

```
ORDER BY([Products];[Products]Type;<;[Products]Price;<)
```

## Example 7

The following line orders the selection of [Products] by type in ascending order and by price in descending order:

```
ORDER BY([Products];[Products]Type;>;[Products]Price;<)
```

## Example 8

The following line orders the selection of [Products] by type in descending order and by price in ascending order:

```
ORDER BY([Products];[Products]Type;<;[Products]Price;>)
```

## Example 9

The following line performs an indexed sort if [Products]Name is indexed:

```
ORDER BY([Products];[Products]Name;>)
```

## Example 10

The following line orders the selection of [Products] by name in ascending order:

```
ORDER BY([Products];[Products]Name)
```

## Example 11

The following line performs a sequential sort, whether or not the fields are indexed:

```
ORDER BY([Products];[Products]Type;>;[Products]Price;>)
```

## Example 12

The following line performs a sequential sort using a related field:

```
ORDER BY([Invoices];[Companies]Name;>) ` Invoices are sorted alphabetically on the Company name field
```

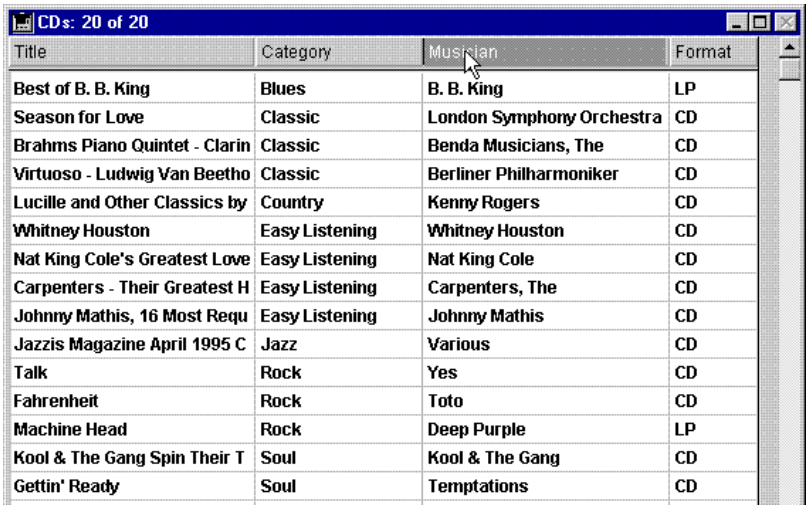
## Example 13

The following example carries out an indexed sort on two levels if a [Contacts]LastName + [Contacts]FirstName composite index has been specified in the database:

```
ORDER BY([Contacts];[Contacts]LastName;>;[Contacts]FirstName;>)
```

## Example 14

In an Output form displayed in the Application environment, you allow the users to order a column in ascending order by simply clicking in the column header.
If the user holds the **Shift** key down while clicking in other column headers, the sort is performed on several levels:




| Title | Category | Musician | Format |
|---|---|---|---|
| Best of B. B. King | Blues | B. B. King | LP |
| Season for Love | Classic | London Symphony Orchestra | CD |
| Brahms Piano Quintet - Clarin | Classic | Benda Musicians, The | CD |
| Virtuoso - Ludwig Van Beetho | Classic | Berliner Philharmoniker | CD |
| Lucille and Other Classics by | Country | Kenny Rogers | CD |
| Whitney Houston | Easy Listening | Whitney Houston | CD |
| Nat King Cole's Greatest Love | Easy Listening | Nat King Cole | CD |
| Carpenters - Their Greatest H | Easy Listening | Carpenters, The | CD |
| Johnny Mathis, 16 Most Requ | Easy Listening | Johnny Mathis | CD |
| Jazzis Magazine April 1995 C | Jazz | Various | CD |
| Talk | Rock | Yes | CD |
| Fahrenheit | Rock | Toto | CD |
| Machine Head | Rock | Deep Purple | LP |
| Kool & The Gang Spin Their T | Soul | Kool & The Gang | CD |
| Gettin' Ready | Soul | Temptations | CD |

Each column header contains a highlight button attached with the following object method:

```
MULTILEVEL(->[CDs]Title) `Title column header button
```

Each button calls the MULTILEVEL project method with a pointer to the corresponding column field. The MULTILEVEL project method is the following:

```
 ` MULTILEVEL Project Method
 ` MULTILEVEL (Pointer)
 ` MULTILEVEL (->[Table]Field)

C_POINTER($1)   `Sort level (field)
C_LONGINT($lLevelNb)

 `Getting sorting levels
If(Not(Shift down)) `Simple sort (one-level)
   ARRAY POINTER(aPtrSortField;1)
   aPtrSortField{1}:=$1
Else
   $lLevelNb:=Find in array(aPtrSortField;$1) `Is this field already sorted?
   If($lLevelNb<0) `If not
      INSERT IN ARRAY(aPtrSortField;Size of array(aPtrSortField)+1;1)
      aPtrSortField{Size of array(aPtrSortField)}:=$1
   End if
End if
 `Performing the sort
$lLevelNb:=Size of array(aPtrSortField)
If($lLevelNb>0) `There is at least one order level
   For($i;1;$lLevelNb)
      ORDER BY([CDs];(aPtrSortField{$i})->;>;*) `Building sort definition
   End for
   ORDER BY([CDs]) `No * ends the sort definition and starts the actual sort operation
End if
```

# ORDER BY FORMULA

ORDER BY FORMULA ( aTable ; expression {; > or <}{; expression2 ; > or <2 ; ... ; expressionN ; > or <N} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table for which to order selected records |
| expression | Expression | ➡ | Expression on which to set the order for each level (can be of type Alphanumeric, Real, Integer, Long Integer, Date, Time or Boolean) |
| > or < | Operator | ➡ | Ordering direction for each level: > to order in ascending order, or < to order in descending order |

## Description

**ORDER BY FORMULA** sorts (reorders) the records of the current selection of *aTable* for the current process. After the sort has been completed, the new first record of the selection becomes the current record.

Note that you must specify *aTable*. You cannot use a default table.

You can sort the selection on one level or on several levels. For each sort level, you specify a expression in *expression* and the sorting order in *> or <*. If you pass the “greater than” symbol (>), the order is ascending. If you pass the “less than” symbol (<), the order is descending. If you do not specify the sorting order, ascending order is the default.

The parameter *expression* can be of type Alphanumeric, Real, Integer, Long Integer, Date, Time or Boolean.

No matter what way a sort has been defined, if the actual sort operation is going to take some time to be performed, 4D automatically displays a message containing a progress thermometer. These messages can be turned on and off by using the **MESSAGES ON** and **MESSAGES OFF** commands. If the progress thermometer is displayed, the user can click the Stop button to interrupt the sort. If the sort is completed, OK is set to 1. Otherwise, if the sort is interrupted, OK is set to 0 (zero).

**4D Server:** Beginning with version 11 of 4D Server, this command is executed on the server, which optimizes its execution. Note that when variables are called directly in the *expression*, the sort is calculated with the value of the variable on the client machine.

On the other hand, this principle does not apply for formulas using methods that, themselves, call variables (the values of the variables are evaluated on the server). In this context, it may be advisable to use the "Execute on server" method attribute that allows a method to be executed on the server while passing parameters (variables) to it (see the Design Reference manual).

In previous versions of 4D Server, this command was executed on the client machines. For compatibility's sake, this functioning is maintained in databases converted to version 11. A compatibility preference and a selector of the **SET DATABASE PARAMETER** command can nevertheless be used to adopt the functioning of version 11 (execution on the server) in these databases.

## Example

This example orders the records of the [People] table in descending order, based on the length of each person’s last name. The record for the person with the longest last name will be first in the current selection:

```
ORDER BY FORMULA([People];Length([People]Last Name);<)
```

# QUERY

QUERY ( {aTable }{;}{ queryArgument {; *}} )

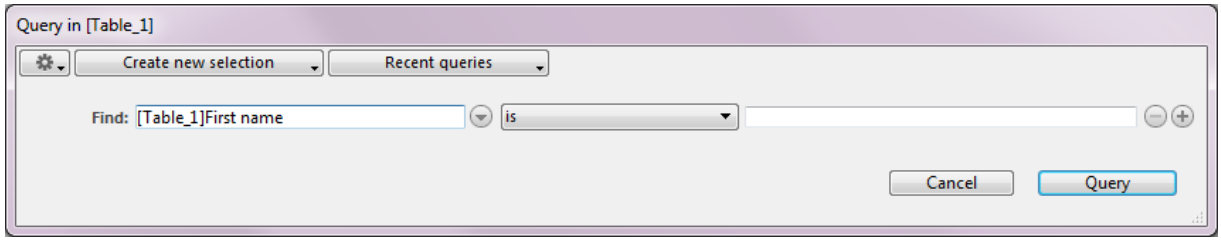
| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return a selection of records, or Default table, if omitted |
| queryArgument | Expression | ⇒ | Query argument |
| * | Operator | ⇒ | Continue query flag |

## Description

**QUERY** looks for records matching the criteria specified in *queryArgument* and returns a selection of records for *aTable*. **QUERY** changes the current selection of *aTable* for the current process and makes the first record of the new selection the current record.

If the *aTable* parameter is omitted, the command applies to the default table. If no default table has been set, an error occurs.

If you do not specify *queryArgument* or the * parameters, **QUERY** displays the Query editor for *aTable* (except when it is the last row of a multiple query, see example 2):




For more information about using the Query Editor, refer to the 4D Design Reference manual.

The user builds the query, then clicks the Query button or chooses Query in selection to perform the query. If the query is performed without interruption, the OK variable is set to 1. If the user clicks Cancel, the **QUERY** terminates with no query actually performed, and sets the OK variable to *0* (zero).

## Example 1

The following line displays the Query editor for the [Products] table:

```
QUERY([Products])
```

## Example 2

The following line displays the Query editor for the default table (if it has been set)

```
QUERY
```

If you specify the *queryArgument* parameter, the standard Query editor is not presented and the query is defined programmatically. For simple queries (search on only one field) you call **QUERY** once with *queryArgument*. For multiple queries (search on multiple fields or with multiple conditions), you call **QUERY** as many times as necessary with *queryArgument*, and you specify the optional * parameter, except for the last **QUERY** call, which starts the actual query operation. The *queryArgument* parameter is described further in this section.

## Example 3

The following line looks for the [People] whose name starts with an “a”:

```
QUERY([People];[People]Last name="a@")
```

## Example 4

The following line looks for the [People] whose name starts with “a” or “b”:

```
QUERY([People];[People]Name="a@";*) ` * indicates that there are further search criteria
QUERY([People];|;[People]Name="b@") ` No * ends the query definition and starts the actual query operation
```

**Note:** The interpretation of @ characters in queries can be modified via an option in the Preferences. For more information, please refer to the **Comparison Operators** section.

## Specifying the Query Argument

The *queryArgument* parameter uses the following syntax:

*{ conjunction ; } field comparator value*

The conjunction is used to join **QUERY** calls when defining multiple queries. The conjunctions available are the same as those in the Query editor:

| Conjunction | Symbol to use with QUERY |
|---|---|
| AND | & |
| OR | \| |
| Except | # |

The *conjunction* is optional and not used for the first **QUERY** call of a multiple query, or if the query is a simple query. If you omit it within a multiply query, AND (&) is used by default.
The *field* is the field to query. The *field* may belong to another table if it belongs to a One table related to *aTable* with an automatic or manual relation.

The *comparator* is the comparison that is made between *field* and *value*. The *comparator* is one of the symbols shown here:

| Comparison | Symbol to use with QUERY |
|---|---|
| Equal to | = |
| Not equal to | # |
| Less than | < |
| Greater than | > |
| Less than or equal to | <= |
| Greater than or equal to | >= |
| Contains keyword | % |

**Note:** It is also possible to specify the comparison operator as an alphanumeric expression instead of a symbol. In this case, it is mandatory to use semi-colons in order to separate the items of the query string. This means that it is possible, for example, to create configurable query sequences by varying the comparison operator, or to build custom user query interfaces. Please refer to example 19.

The *value* is the data against which *field* will be compared. The value can be any expression that evaluates to the same data type as *field*. The value is evaluated once, at the beginning of the query. The value is not evaluated for each record. To query for a string contained in a string (a “contains” query), use the wildcard symbol (@) in *value* to isolate the string to be searched for as shown in this example "@Smith@". Note that in this case, the search only partially benefits from the index (compactness of data storage).

Searching by keywords is only available with Alpha or Text type fields. For more information about this type of query, please refer to the **Comparison Operators** section.

Here are the rules for building multiple queries:

- The first query argument must not contain a conjunction.
- Each successive query argument can begin with a conjunction. If you omit it, the AND (&) operator is used by default.
- The first query and every other query, except the last, must use the * parameter.
- To perform the query, do not specify the * parameter in the last **QUERY** command. Alternatively, you may execute the **QUERY** command without any parameters other than the table (the Query editor is not shown; instead, the multiple query you just defined is performed).

**Note:** Each table maintains its own current built query. This means that you can create multiple built queries simultaneously, one for each table. You must use the *aTable* parameter or set the default table to specify which table to use.

No matter which way a query has been defined:

- If the actual query operation is going to take some time to be performed, 4D automatically displays a message containing a progress thermometer. These messages can be turned on and off by using the **MESSAGES ON** and **MESSAGES OFF** commands. If the progress thermometer is displayed, the user can click on the Stop button to

interrupt the query. If the query is completed, OK is set to 1. Otherwise, if the query is interrupted, OK is set to 0 (zero).
- If any indexed fields are specified, the query is optimized every time that it is possible (indexed fields are searched first) resulting in a query that takes the least amount of time possible. The command makes use of composite indexes for queries using the AND (&).

## Example 5

The following command finds the records for all the people named Smith:

```
QUERY([People];[People]Last Name="Smith")
```

**Note:** If the Last Name field were indexed, the **QUERY** command would automatically use the index for a fast query.

**Reminder:** This query will find records like "Smith", "smith","SMITH", etc. To distinguish lowercase from uppercase, perform additional queries using the character codes.

## Example 6

The following example finds the records for all people named John Smith. The Last Name field is indexed. The First Name field is not indexed.

```
QUERY([People];[People]Last Name="smith";*) ` Find every person named Smith
QUERY([People];&;[People]First Name="john") ` with John as first name
```

When the query is performed, it quickly does an indexed search on Last Name and reduces the selection of records to those of people named Smith. The query then sequentially searches on First Name in this selection of records.

**Note:** This query is particularly optimized if the database contains a composite index including the *[People]Last Name+ [People]First Name* fields. In this case, the command takes advantage of the index and the query is completely indexed.

## Example 7

The following example finds the records of people named Smith or Jones. The Last Name field is indexed.

```
QUERY([People];[People]Last Name="smith";*) ` Find every person named Smith…
QUERY([People];|;[People]Last Name="jones") ` …or Jones
```

The **QUERY** command uses the Last Name index for both queries. The two queries are performed, and their results put into internal sets that are eventually combined using a union.

## Example 8

The following example finds the records for people who do not have a company name. It does this by finding entries with empty fields (the empty string).

```
QUERY([People];[People]Company="") ` Find every person with no company
```

## Example 9

The following example finds the record for every person whose last name is Smith and who works for a company based in New York. The second query uses a field from another table. This query can be done because the [People] table is related to the [Company] table with a many to one relation:

```
QUERY([People];[People]Last Name="smith";*) ` Find every person named Smith…
QUERY([People];&;[Company]State="NY") ` … who works for a company based in NY
```

## Example 10

The following example finds the record for every person whose name falls between A (included) and M (included):

```
QUERY([People];[People]Name<"n") ` Find every person from A to M
```

## Example 11

The following example finds the records for all the people living in the San Francisco or Los Angeles areas (ZIP codes beginning with 94 or 90):

```
QUERY([People];[People]ZIP Code ="94@";*) ` Find every person in the SF…
QUERY([People];|;[People]ZIP Code ="90@") ` …or Los Angeles areas
```

## Example 12

Searching by keyword: the following example searches the [Products] table for records where the Description field contains the word “easy”:

```
QUERY([Products];[Products]Description%"easy")
  ` Find products whose description contains the keyword easy
```

## Example 13

The following example finds the record that matches the invoice reference entered in the request dialog box:

```
vFind:=Request("Find invoice reference:") ` Get an invoice reference from the user
If(OK=1) ` If the user pressed OK
   QUERY([Invoice];[Invoice]Ref=vFind) ` Find the invoice reference that matches vFind
End if
```

## Example 14

The following example finds the records for the invoices entered in 1996. It does this by finding all records entered after 12/31/95 and before 1/1/97:

```
QUERY([Invoice];[Invoice]In Date>!12/31/95!;*) ` Find invoices after 12/31/95…
QUERY([Invoice];&;[Invoice]In Date<!1/1/97!) ` and before 1/1/97
```

## Example 15

The following example finds the record for each employee whose salary is between $10,000 and $50,000. The query includes the employees who make $10,000, but excludes those who make $50,000:

```
QUERY([Employee];[Employee]Salary >=10000;*) ` Find employees who make between…
QUERY([Employee];&;[Employee]Salary <50000) ` …$10,000 and $50,000
```

## Example 16

The following example finds the records for the employees in the marketing department who have salaries over $20,000. The Salary field is queried first because it is indexed. Notice that the second query uses a field from another table. It can do this because the [Dept] table is related to the [Employee] table with an automatic many to one relation:

```
QUERY([Employee];[Employee]Salary >20000;*) ` Find employees with salaries over $20,000 and…
QUERY([Employee];&;[Dept]Name="marketing") ` …who are in the marketing department
```

## Example 17

Given three tables related by Many-to-One relations: [City] -> [Department] -> [Region]. The following query finds all the regions with cities whose names begin with "Saint":

```
QUERY([Region];[City]Name="Saint@") ` Find all the regions with cities beginning with "Saint"
```

## Example 18

The following example queries for information that was entered into the variable *myVar*.

```
QUERY([Laws];[Laws]Text =myVar) ` Find all laws that match myVar
```

The query could have many different results, depending on the value of *myVar*. The query will also be performed differently. For example:

- If *myVar* equals *"Copyright@"*, the selection contains all laws with texts beginning with Copyright.
- If *myVar* equals *"@Copyright@"*, the selection contains all laws with texts containing at least one occurrence of Copyright.

## Example 19

The following example adds or does not add lines to a complex query depending on the value of the variables. This way, only valid criteria are taken into account for the query:

```
QUERY([Invoice];[Invoice]Paid=False;*)
If($city#"") ` if a city name has been specified
   QUERY([Invoice];[Invoice]Delivery_city=$city;*)
End if
If($zipcode#"") ` If a zip code has been specified
   QUERY([Invoice];[Invoice]ZipCode=$zipcode;*)
End if
QUERY([Invoice]) ` Execution of query on the criteria
```

## Example 20

This example illustrates the use of a comparison operator as an alphanumeric expression. The value of the comparison operator is specified using a pop-up menu placed in a custom query dialog box:

```
C_TEXT($oper)
$oper:=_popup_operator{_popup_operator} `$oper equals for example "#" or "="
If(OK=1)
   QUERY(Invoice];[Invoice]Amount;$oper;$amount)
End if
```

## Example 21

Using picture keyword indexes can greatly increase the speed of your applications.

```
QUERY([PICTURES];[PICTURES]Photos %"cats") // look for photos associated with the "cats" keyword
```

## System variables and sets

If the query is carried out correctly, the OK system variable is set to 1.
The OK variable is set to 0 if:

- the user clicks on the **Cancel**/**Stop** button,
- in 'query and lock' mode (see the **SET QUERY AND LOCK** command), the query has found at least one locked record. In this case as well, the LockedSet system set is updated.

# QUERY BY ATTRIBUTE

QUERY BY ATTRIBUTE ( {aTable}{;}{conjOp ;} objectField ; attributePath ; queryOp ; value {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table for which to return a selection of records, or Default table if omitted |
| conjOp | Operator | ➡ | Conjunction operator to use to join multiple queries (if any) |
| objectField | Field | ➡ | Object field to query attributes |
| attributePath | String | ➡ | Name or path of attribute |
| queryOp | Operator, String | ➡ | Query operator (comparator) |
| value | Text, Number, Date, Time | ➡ | Value to compare |
| * | Operator | ➡ | Continue query flag |

## Description

**QUERY BY ATTRIBUTE** looks for records matching the query string defined using the *objectField*, *attributePath*, *queryOp* and *value* parameters, and returns a selection of records for *aTable*.

**Note:** For more information on Object fields (new in 4D v15), please refer to the section of the *Design Reference* manual.

**QUERY BY ATTRIBUTE** changes the current selection of *aTable* for the current process and makes the first record of the new selection the current record. If the *aTable* parameter is omitted, the command applies to the default table. If no default table has been set, an error occurs.

The optional *conjOp* parameter is used to join **QUERY BY ATTRIBUTE** calls when defining multiple queries. The conjunction operators available are the same as the ones for the **QUERY** command:

| Conjunction | Symbol to use with QUERY BY ATTRIBUTE |
|---|---|
| AND | & |
| OR | \| |
| Except | # |

The *conjOp* parameter is not used for the first **QUERY BY ATTRIBUTE** call of a multiple query, or if the query is a simple query. If you omit it within a multiple query, the AND (&) operator is used by default.

In *objectField*, pass the Object field whose attribute(s) you want to query. If it belongs to a One table related to *aTable* with an automatic or manual relation, the *objectField* may belong to another table.
**QUERY BY ATTRIBUTE** supports 4D Write Pro custom attributes when documents are stored in Object fields. For more information about this point, please refer to the **Storing 4D Write Pro documents in 4D Object fields** section.

In *attributePath*, pass the path of the attribute whose values you want to compare for each record, for example "children.girls.age". If you pass a single name, for example "place", you designate the corresponding attribute found at the first level of the object field.
If an attribute "x" is an array, **QUERY BY ATTRIBUTE** will search records which contain an attribute "x" in which at least one element matches the criteria. To search in array attributes, it is necessary to indicate to the **QUERY BY ATTRIBUTE** command that attribute "x" is an array by appending "[]" to its name in *attributePath* (see example 3).

**Notes:**

- Keep in mind that attribute names are case-sensitive: you can have different "MyAtt" and "myAtt" attribute names in the same record.
- Attribute names are trimmed to eliminate extra spaces. For example, " my first attribute .my second attribute " is interpreted as "my first attribute.my second attribute".

The *queryOp* parameter is the comparison operator that is applied between *objectField* and *value*. You can pass one of the symbols shown here:

| Comparison | Symbol to use with QUERY BY ATTRIBUTE |
|---|---|
| Equal to | = |
| Not equal to | # |
| Less than | < |
| Greater than | > |
| Less than or equal to | <= |
| Greater than or equal to | >= |

**Note:** It is also possible to specify the comparison operator as an text expression instead of a symbol. See the **QUERY** command description for more information.

*value* is the data against which *attributePath* will be compared. The value can be any expression that evaluates to the same data type as *attributePath*. The value is evaluated once, at the beginning of the query. The value is not evaluated for each record. To query for a string contained within a string (a "contains" query), use the wildcard symbol (@) in *value* to isolate the string to be searched for as shown in this example: "@Smith@". Note that in this case, the search only partially benefits from the index (compactness of data storage).

Here is the structure of a query by attribute:

```
QUERY BY ATTRIBUTE([Table] ;[Table]ObjectField ;"attribute1.attribute2";=;value)
```

**Note:** An implicit criteria for all operators (except #) is that the Object field contains an attribute. However, for the # operator, it can be undefined (see below).

### Using the # operator (support for Null values)

Queries by attribute using the "#" operator can have different results depending on whether or not the property is checked for the object field:

- **Map NULL values to blank values** property checked (default option, recommended in most cases).
  In this case, the "#" operator should be seen as selecting records where "no attribute" of the field contains the value searched for. In this context, 4D considers in a similar manner:
  - fields where the value of the attribute is different from the value searched for,
  - fields where the attribute is not present (or contains a Null value).

  For example, the following query returns records for people who have a dog whose name is not Rex, as well as records for people who do not have a dog, or who have a dog with no name:

  ```
  QUERY BY ATTRIBUTE([People];[People]Animals;"dog.name";#;"Rex")
  ```

  Another example: this query will return all records for which *[Table]ObjectField* contains an object which contains an *attribute1* attribute which is itself an object containing an *attribute2* attribute whose value is not *value*, as well as records where the object field does not contain *attribute1* or *attribute*2):

  ```
  QUERY BY ATTRIBUTE([Table];[Table]ObjectField;"attribute1.attribute2";#;value)
  ```

  This principle also applies to array attributes. For example:

  ```
  QUERY BY ATTRIBUTE([People];[People]OB_Field;"locations[].city";#;"paris")
  ```

  This query will return records for people who do not have any address in Paris.

  To specifically obtain records where the attribute is undefined, you can use an empty object (see example 2). Note however that searching for NULL values in array elements is not supported.

- **Map NULL values to blank values** property unchecked ("SQL" mode).
  In this case, undefined attributes (attributes not present in the field or whose value is Null) are not considered as equivalent to blank values by default. As a result, queries of the type "attribute A is different from attribute B" will not return records where attribute A is undefined.
  To use the same example as above, when the **Map NULL values to blank values** option is not checked for the [People]Animals field, the following query will only return records for people who have a dog whose "name" attribute does not contain "Rex". Records for people who do not have a dog, or who have a dog with no name will not be returned in this case.

  ```
  QUERY BY ATTRIBUTE([People];[People]Animals;"dog.name";#;"Rex")
  ```

  This operation, closer to the SQL logic, is reserved for specific needs.

### Building multiple queries

Here are the rules for building multiple queries by attribute:

- The first query argument must not contain a conjunction.
- Each successive query argument can begin with a conjunction. If you omit it, the AND (&) operator is used by default.
- The first query and every other query, except the last, must use the * parameter.

- **QUERY BY ATTRIBUTE** can be mixed with **QUERY** commands (see example).
- To perform the query, do not specify the * parameter in the last **QUERY BY ATTRIBUTE** command. Alternatively, you can execute the **QUERY** command without any parameters other than the table.

**Note:** Each table maintains its own currently-built query. This means that you can create multiple queries simultaneously, one for each table.

No matter which way a query has been defined:

- If the actual query operation is going to take some time to be performed, 4D automatically displays a message containing a progress thermometer. These messages can be turned on and off by using the **MESSAGES ON** and **MESSAGES OFF** commands. If a progress thermometer is displayed, the user can click on the Stop button to interrupt the query. If the query is completed, OK is set to 1. Otherwise, if the query is interrupted, OK is set to 0 (zero).
- If any indexed object fields are specified, the query is optimized every time that it is possible (indexed fields are searched first) resulting in a query that takes the least amount of time possible.

### Date values in the object

Dates are stored in objects according to database settings; by default, the time zone is taken into account (see the JSON use local time selector in the **SET DATABASE PARAMETER** command).

```
!1973-05-22! -> "1973-05-21T23:00:00.000Z"
```

This setting is also taken into account during queries, so you do not have to worry about it if you always use your database at the same place and if settings are the same on all machines that access the data. In this case, the following query will correctly return records whose Birthday attribute equals !1973-05-22! (saved as "1973-05-21T23:00:00.00Z"):

```
QUERY BY ATTRIBUTE([Persons];[Persons]OB_Info;"Birthday";=;!1973-05-22!)
```

If you do not want to use the GMT settings, you can modify these settings using the following instruction:

```
SET DATABASE PARAMETER(JSON use local time;0)
```

Keep in mind that the scope of this setting is the process only. If you execute this instruction, then October 1st, 1965 will be stored "1965-10-01T00:00:00.000Z" but you will need to set the same parameter before launching your queries:

```
SET DATABASE PARAMETER(JSON use local time;0)
QUERY BY ATTRIBUTE([Persons];[Persons]OB_Info;"Birthday";=;!1976-11-27!)
```

### Using the virtual length property

You can use the virtual "length" property with this command. This property is available automatically for all array type attributes and returns the size of the array, i.e. the number of elements it contains. It can be used in the context of executing the **QUERY BY ATTRIBUTE** command (see example 4).

## Example 1

In this example, the "age" attribute is either a string or an integer and we want to find people whose age is between 20 and 29. The first two lines query the attribute as an integer (>=20 and < 30) and the last ones query the field as a string (starts with "2" but is different from "2".)

```
QUERY BY ATTRIBUTE([Persons];[Persons]OB_Info;"age";>=;20;*)
QUERY BY ATTRIBUTE([Persons]; & ;[Persons]OB_Info;"age";<;30;*)
QUERY BY ATTRIBUTE([Persons];|;[Persons]OB_Info;"age";=;"2@";*)
QUERY BY ATTRIBUTE([Persons]; & ;[Persons]OB_Info;"age";#;"2") //no final * to launch execution
```

## Example 2

The **QUERY BY ATTRIBUTE** command can be used to find records where certain attributes are defined (or are not defined). To do this, you have to use an empty object.

```
 //Find records where e-mail is defined in the object field
C_OBJECT($undefined)
QUERY BY ATTRIBUTE([Persons];[Persons]Info;"e-mail";#;$undefined)
```

```
//Find records where zip code is NOT defined in the object field
C_OBJECT($undefined)
QUERY BY ATTRIBUTE([Persons];[Persons]Info;"zip code";=;$undefined)
```

**Note:** This specific syntax is not supported with array type attributes. Searching for NULL values in array elements will give invalid results.

## Example 3

You want to search a field containing array attributes. With the following two records:

*Record1:*
[People]name: "martin"
[People]OB_Field:
```
   "locations" : [ {
            "kind":"office",
            "city":"paris"
         } ]
```
*Record2:*
[People]name: "smith"
[People]OB_Field:
```
   "locations" : [ {
            "kind":"home",
            "city":"lyon"
         } , {
            "kind":"office",
            "city":"paris"
         } ]
```
... **QUERY BY ATTRIBUTE** will find people with a location in "paris" using this statement:

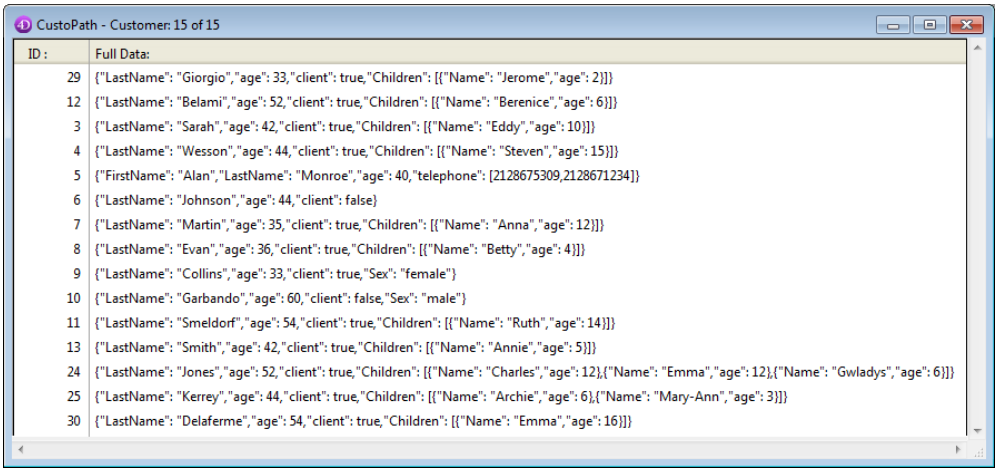
```
//flag the array attribute with "[]" syntax
QUERY BY ATTRIBUTE([People];[People]OB_Field;"locations[].city";=;"paris")
//selects "martin" and "smith"
```

**Note:** If you defined several criteria on the same array attribute, the matched criteria will not necessarily apply to the same array element. In the following example, the query returns "smith" because it has a "locations" element whose "kind" is "home" and a "locations" element whose "city" is "paris", even if it's not the same element:

```
QUERY BY ATTRIBUTE([People];[People]OB_Field;"locations[].kind";=;"home";*)
QUERY BY ATTRIBUTE([People]; & ;[People]OB_Field;"locations[].city";=;"paris")
//selects "smith"
```

## Example 4

This example illustrates the use of the virtual "length" property. Your database has a [Customer]full_Data object field with the following data:



You want to get the records for any customers who have two or more children. To do this, you can write:

```
QUERY BY ATTRIBUTE([Customer];[Customer]full_Data;"Children.length";>=;2)
```

## System variables and sets

If the query is carried out correctly, the OK system variable is set to 1.
The OK variable is set to 0 if:

- the user clicks on the **Cancel**/**Stop** button,
- in 'query and lock' mode (see the **SET QUERY AND LOCK** command), the query has found at least one locked record. In this case as well, the LockedSet system set is updated.

# QUERY BY EXAMPLE

QUERY BY EXAMPLE ( {aTable}{;}{*} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return a selection of records, or Default table, if omitted |
| * | Operator | ⇒ | If passed, the scrolling bar will not be displayed |

## Description

**QUERY BY EXAMPLE** performs the same action as the Query by Example menu command in the Design environment. It displays the current input form as a query window. **QUERY BY EXAMPLE** queries *aTable* for the data that the user enters into the query window. The form must contain the fields that you want the user to be able to query. The query is optimized; indexed fields are automatically used to optimize the query.

See the 4D Design Reference manual for information about using the Query by Example menu command in the Design environment.

## Example

The method in this example displays the MyQuery form to the user. If the user accepts the form and performs the query (that is, if the OK system variable is set to 1), the records that meet the query criteria are displayed:

```
FORM SET INPUT([People];"MyQuery") ` Switch to query form
QUERY BY EXAMPLE([People]) ` Display form and perform query
If(OK=1) ` If the user performed the query
   DISPLAY SELECTION([People]) ` Display the records
End if
```

## System variables and sets

If the user clicks the Accept button or presses the Enter key, the OK system variable is set to 1 and the query is performed. If the user clicks the Cancel button or presses the “cancel” key combination, the OK system variable is set to 0 and the query is canceled.

# QUERY BY FORMULA

QUERY BY FORMULA ( aTable {; queryFormula} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return a selection of records |
| queryFormula | Boolean | ⇒ | Query formula |

## Description

**QUERY BY FORMULA** looks for records in *aTable*. It changes the current selection of *aTable* for the current process and makes the first record of the new selection the current record.

**QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** work exactly the same way, except that **QUERY BY FORMULA** queries every record in the entire table and **QUERY SELECTION BY FORMULA** queries only the records in the current selection.

Both commands apply *queryFormula* to each record in the table or selection. The *queryFormula* is a Boolean expression that must evaluate to either TRUE or FALSE. If *queryFormula* evaluates as TRUE, the record is included in the new selection.

The *queryFormula* may be simple, perhaps comparing a field to a value; or it may be complex, perhaps performing a calculation or even evaluating information in a related table. The *queryFormula* can be a 4D function (command), or a function (method) or expression you have created. You can use wildcards (@) in *queryFormula* when working with Alpha or text fields as well as the "contains" (%) operator for keyword queries. For more information, please refer to the description of the **QUERY** command.

If *queryFormula* is omitted, 4D displays the query dialog box (the user can add a line of formula by **Alt+clicking** on the sur le **[+]** button).

When the query is complete, the first record of the new selection is loaded from disk and made the current record.

These commands are optimized and can more particularly take advantage of indexes. When the type of query allows it, these commands execute queries equivalent to the **QUERY** command. For example, the statement **QUERY BY FORMULA**([mytable]; [mytable]myfield=value) will be executed just like **QUERY**([mytable]; [mytable]myfield=value), which will allow the use of indexes. 4D can also optimize queries containing parts that cannot be optimized, by first executing the optimized parts and then combining the results with the rest of the query. For example, the statement **QUERY BY FORMULA**[mytable];Length(myfield)=value) will not be optimized. On the other hand, **QUERY BY FORMULA**([mytable];Length(myfield)=value1 | myfield=value2) will be partially optimized.

These commands by default carry out "joins" like SQL when you compare fields from different tables. This means that it is not necessary for a structural automatic relation to exist between the tables. For example, you can execute a statement of the type **QUERY BY FORMULA([Table_A];([Table_A]field_X = [Table_B]field_Y) & ([Table_B]field_Y = "abc"))** (see example 3). The first part of the formula (*[Table_A]field_X = [Table_B]field_Y*) establishes the join between the two fields and the second part (*[Table_B]field_Y = "abc"*) defines the search criteria (at least one criterion must be set).
If they exist, the relations between the tables are not used as a rule. However, these commands will use automatic relations in the following cases:

- If the formula cannot be broken down into elements of the *{ field ; comparator ; value}* form
- If two fields of the same table are compared.

**Note:** For compatibility reasons, it is possible to deactivate the joins mechanism, either globally via the database Preference (converted databases only) or per process using the **SET DATABASE PARAMETER** command.

**4D Server:** Beginning with version 11 of 4D Server, these commands are run on the server, which optimizes their execution. Keep in mind that when variables are called directly in *queryFormula*, the query is calculated with the value of the variables on the client machine. For example, the statement **QUERY BY FORMULA**([mytable];[mytable]myfield=myvariable) will be run on the server but with the contents of the myvariable variable of the client machine.

On the other hand, this principle is not applied for formulas using methods that, themselves, call variables (the values of the variables are evaluated on the server). In this context, it may be advisable to use the "Execute on server" method attribute, which allows the method to be executed on the server while passing parameters (variables) to it (see the Design Reference manual).

In previous versions of 4D Server, these commands were executed on client machines. For compatibility's sake, this functioning is maintained for databases converted to version 11. A compatibility preference and a selector of the **SET**

**DATABASE PARAMETER** command can nevertheless be used to adopt the functioning of version 11 (execution on the server) in converted databases

## Example 1

This example finds the records for all invoices that were entered in December of any year. It does this by applying the **Month of** function to each record. This query could not be performed any other way without creating a separate field for the month:

```
QUERY BY FORMULA([Invoice];Month of([Invoice]Entered)=12) ` Find the invoices entered in December
```

## Example 2

This example finds records for all the people who have names with more than ten characters:

```
QUERY BY FORMULA([People];Length([People]Name)>10) ` Find names longer than ten characters
```

## Example 3

This example activates SQL joins for a specific query by formula:

```
$currentVal:=Get database parameter(QUERY BY FORMULA Joins)
SET DATABASE PARAMETER(QUERY BY FORMULA Joins;2) `Activate SQL joins
 `Query all the lines of "ACME" client invoices even though the tables are not related
QUERY BY FORMULA([invoice_line];([invoice_line]invoice_id=[invoice]id&[invoice]client="ACME"))
SET DATABASE PARAMETER(QUERY BY FORMULA Joins;$currentVal) `We re-establish the current settings
```

# QUERY SELECTION

QUERY SELECTION ( {aTable }{;}{ queryArgument {; *}} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return a selection of records, or Default table, if omitted |
| queryArgument | Expression | ⇒ | Query argument |
| * | Operator | ⇒ | Continue query flag |

## Description

**QUERY SELECTION** looks for records in *aTable*.The **QUERY SELECTION** command changes the current selection of *table* for the current process and makes the first record of the new selection the current record.

**QUERY SELECTION** works and performs the same actions as **QUERY**. The difference between the two commands is the scope of the query:

- **QUERY** looks for records among all the records in the table.
- **QUERY SELECTION** looks for records among the records currently selected in the table.

For more information, see the description of the **QUERY** command.

The **QUERY SELECTION** command is useful when a query cannot be defined using a sequence of **QUERY** calls joined with the * parameter. Typically, it is the case when you want to query a current selection that does not result from a previous query, but from a command such as **USE SET**.

## Example

You want to query the records that have been previously highlighted by the user in a list form. You can write:

```
USE SET("UserSet") //replace the current selection with the highlighted records
QUERY SELECTION([Company];[Company]City="New York City";*)
QUERY SELECTION([Company]Type Business="Stock Exchange")
```

You will find all companies located in New York City, with a Stock Exchange activity, among the initial user selection.

# QUERY SELECTION BY ATTRIBUTE

QUERY SELECTION BY ATTRIBUTE ( {aTable}{;}{conjOp ;} objectField ; attributePath ; queryOp ; value {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table for which to return a selection of records, or Default table if omitted |
| conjOp | Operator | ➡ | Conjunction operator to use to join multiple queries (if any) |
| objectField | Field | ➡ | Object field to query attributes |
| attributePath | String | ➡ | Name or path of attribute |
| queryOp | Operator, String | ➡ | Query operator (comparator) |
| value | Text, Number, Date, Time | ➡ | Value to compare |
| * | Operator | ➡ | Continue query flag |

## Description

**QUERY SELECTION BY ATTRIBUTE** works and performs the same actions as **QUERY BY ATTRIBUTE**. The difference between these two commands is the scope of the query:

- **QUERY BY ATTRIBUTE** looks for records among all the records in the table.
- **QUERY SELECTION BY ATTRIBUTE** looks for records among the records currently selected in the table.

**QUERY SELECTION BY ATTRIBUTE** looks for records in *aTable*. The **QUERY SELECTION BY ATTRIBUTE** command changes the current selection of *aTable* for the current process and makes the first record of the new selection the current record.

For more information, see the description of the **QUERY BY ATTRIBUTE** command.

The **QUERY SELECTION BY ATTRIBUTE** command is useful when a query cannot be defined using a combination of **QUERY BY ATTRIBUTE** (and even **QUERY**) calls joined with the * parameter. Typically, this is the case when you want to query a current selection that does not result from a previous query, but from a command such as **USE SET**.

## Example

You want to find people with an age between 20 and 30, among the records that were previously highlighted by the user:

```
USE SET("UserSet") //creates a new current selection
QUERY SELECTION BY ATTRIBUTE([People];[People]OB_Info;"age";>;20;*)
QUERY SELECTION BY ATTRIBUTE([People];&;[People]OB_Info;"age";<;30) //triggers the query
```

# QUERY SELECTION BY FORMULA

QUERY SELECTION BY FORMULA ( aTable {; queryFormula} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return a selection of records |
| queryFormula | Boolean | ⇒ | Query formula |

## Description

**QUERY SELECTION BY FORMULA** looks for records in *aTable*. **QUERY SELECTION BY FORMULA** changes the current selection of *aTable* for the current process and makes the first record of the new selection the current record.

**QUERY SELECTION BY FORMULA** performs the same actions as **QUERY BY FORMULA**. The difference between the two commands is the scope of the query:

- **QUERY BY FORMULA** looks for records among all the records in the table.
- **QUERY SELECTION BY FORMULA** looks for records among the records currently selected in the table.

For more information, see the description of the **QUERY BY FORMULA** command.

# QUERY SELECTION WITH ARRAY

QUERY SELECTION WITH ARRAY ( targetField ; array )

| Parameter | Type | | Description |
|---|---|---|---|
| targetField | Field | ⇒ | Field used to compare the values |
| array | Array | ⇒ | Array of searched values |

## Description

The **QUERY SELECTION WITH ARRAY** command searches the table of the field passed as first parameter for the records where the value of *targetField* is equal to at least one of the values of the elements in the *array*. The records found will become the new current selection.

**QUERY SELECTION WITH ARRAY** functions in the same way as **QUERY WITH ARRAY**. The difference between these two commands is the scope of the search:

- **QUERY WITH ARRAY** searches all the records of the table containing *targetField*.
- **QUERY SELECTION WITH ARRAY** only searches the records of the current selection of the table containing *targetField*.

For more information, please refer to the description of the **QUERY WITH ARRAY** command.

# QUERY WITH ARRAY

QUERY WITH ARRAY ( targetField ; array )

| Parameter | Type | | Description |
|---|---|---|---|
| targetField | Field | ⇒ | Field used to compare the values |
| array | Array | ⇒ | Array of the searched values |

## Description

The **QUERY WITH ARRAY** command searches all the records for which the value of *targetField* is equal, at least, to one of the values of the elements in *array*. The records found will become the new current selection.

This command allows you to quickly and simply build a search on multiple values.

**Notes:**

- This command cannot be used with fields of the Picture or BLOB type.
- *targetField* and *array* must be of the same data type. Exception: you can use a Longint array with a field of the Time type.

## Example

The following example allows you to retrieve the records of both French and American clients:

```
ARRAY STRING(2;SearchArray;2)
SearchArray{1}:="FR"
SearchArray{2}:="US"
QUERY WITH ARRAY([Clients]Country;SearchArray)
```

# SET QUERY AND LOCK

SET QUERY AND LOCK ( lock )

| Parameter | Type | | Description |
|---|---|---|---|
| lock | Boolean | ⇒ | True = Lock the records found by queries False = Do not lock records |

## Description

The **SET QUERY AND LOCK** command can be used to request the automatic locking of records found by all queries that follow the calling of this command in the current transaction. This means that the records cannot be modified by a process other than the current process between a query and the handling of results.

By default, the records found by queries are not locked. Pass **True** in the *lock* parameter to activate locking.

It is imperative for this command to be used within a transaction. If it is called outside of this context, an error is generated. This allows for better control of record locking. The records found will stay locked as long as the transaction has not been terminated (whether validated or cancelled). After the transaction is completed, all the records are unlocked.

The records are locked for all the tables in the current transaction.

When a **SET QUERY AND LOCK**(True) statement has been executed, the query commands (for example **QUERY**) adopt a specific functioning if a record that is already locked is found:

- The query is stopped and the system variable OK is set to 0,
- The current selection is cleared,
- The *LockedSet* system set contains the locked record that caused the query to be stopped.

Consequently, in this context it is necessary to test the *LockedSet* set after a fruitless query (current selection empty and/or OK variable set to 0) in order to determine the cause of the failure.

Call **SET QUERY AND LOCK**(False) in order to disable this mechanism afterward.

**SET QUERY AND LOCK** only modifies the behavior for query commands, in other words:

- **QUERY**
- **QUERY SELECTION**
- **QUERY BY EXAMPLE**
- **QUERY BY FORMULA**
- **QUERY BY SQL**
- **QUERY SELECTION BY FORMULA**
- **QUERY SELECTION WITH ARRAY**
- **QUERY WITH ARRAY**
- **QUERY BY ATTRIBUTE**
- **QUERY SELECTION BY ATTRIBUTE**

However, **SET QUERY AND LOCK** does not affect other commands that modify the current selection such as **ALL RECORDS**, **RELATE MANY**, etc.

## Example

In this example, it is not possible to delete a client who would have been passed from category “C” to category “A” in another process between the **QUERY** and **DELETE SELECTION** commands:

```
START TRANSACTION
SET QUERY AND LOCK(True)
QUERY([Customers];[Customers]Category=C)
 `At this moment, the records found are automatically locked for all other processes
DELETE SELECTION([Customers])
SET QUERY AND LOCK(False)
VALIDATE TRANSACTION
```

## Error management

If the command is not called in the context of a transaction, an error is generated.

# SET QUERY DESTINATION

SET QUERY DESTINATION ( destinationType {; destinationObject {; destinationPtr}} )

| Parameter | Type | | Description |
|---|---|---|---|
| destinationType | Longint | ⇒ | 0 = current selection, 1 = set, 2 = named selection, 3 = variable |
| destinationObject | String, Variable | ⇒ | Name of the set, or Name of the named selection, or Variable |
| destinationPtr | Pointer | ⇒ | Pointer to local variable if destinationType=3 |

## Description

**SET QUERY DESTINATION** enables you to tell 4D where to put the result of any subsequent query for the current process.

You specify the type of the destination in the parameter *destinationType*. 4D provides the following predefined constants, found in the "**Queries**" theme:

| Constant | Type | Value |
|---|---|---|
| Into current selection | Longint | 0 |
| Into named selection | Longint | 2 |
| Into set | Longint | 1 |
| Into variable | Longint | 3 |

You specify the destination of the query itself in the optional *destinationObject* parameter according to the following table:

| destinationType parameter | destinationObject parameter |
|---|---|
| 0 (current selection) | You omit the parameter |
| 1 (set) | You pass the name of a set (existing or to be created) |
| 2 (named selection) | You pass the name of a named selection (existing or to be created) |
| 3 (variable) | You pass a numeric variable (existing or to be created) or an empty string "" to use the *destinationPtr* parameter |

With:

```
SET QUERY DESTINATION(Into current selection)
```

The records found by any subsequent query will end up in a new current selection for the table involved by the query.

With:

```
SET QUERY DESTINATION(Into set;"mySet")
```

The records found by any subsequent query will end up in the set *"mySet"*. The current selection and the current record for the table involved by the query are left unchanged.

With:

```
SET QUERY DESTINATION(Into named selection;"myNamedSel")
```

The records found by any subsequent query will end up in the named selection *"myNamedSel"*. The current selection and the current record for the table involved by the query are left unchanged.

**Notes:**

- If the named selection does not exist beforehand, it will be created automatically at the end of the query.
- This command manages named selections like the **CUT NAMED SELECTION** command: only references are kept. Once the named selection is used, it no longer exists.

With:

```
SET QUERY DESTINATION(Into variable;$vlResult)
```

Or:

```
SET QUERY DESTINATION(Into variable;"";->$vlResult)
```

**Note:** This second syntax facilitates the joint use of the command with **GET QUERY DESTINATION**.

The number of records found by any subsequent query will end up in the variable *$vlResult*. The current selection and the current record for the table involved by the query are left unchanged.

**Warning: SET QUERY DESTINATION** affects all subsequent queries made within the current process. REMEMBER to always counterbalance a call to **SET QUERY DESTINATION** (where *destinationType#0*) with a call to **SET QUERY DESTINATION**(0) in order to restore normal query mode.

**SET QUERY DESTINATION** changes the behavior of the query commands only:

- **QUERY**
- **QUERY SELECTION**
- **QUERY BY EXAMPLE**
- **QUERY BY FORMULA**
- **QUERY BY SQL**
- **QUERY SELECTION BY FORMULA**
- **QUERY SELECTION WITH ARRAY**
- **QUERY WITH ARRAY**
- **QUERY BY ATTRIBUTE**
- **QUERY SELECTION BY ATTRIBUTE**

On the other hand, **SET QUERY DESTINATION** does not affect other commands that may change the current selection of a table such as **ALL RECORDS**, **RELATE MANY** and so on.

## Example 1

You create a form that will display the records from a *[Phone Book]* table. You create a Tab Control named *asRolodex* (with the 26 letters of the alphabet) and a subform displaying the *[Phone Book]* records. Choosing one Tab from the Tab Control displays the records whose names start with the corresponding letter.

In your application, the *[Phone Book]* table contains a set of quite static data, so you do not want to (or need to) perform a query each time you select a Tab. In this way, you can save precious database engine time.

To do so, you can redirect your queries into named selections that you reuse as needed. You write the object method of the Tab Control *asRolodex* as follows:

```
  ` asRolodex object method
Case of
   :(Form event=On Load)
  ` Before the form appears on the screen,
  ` initialize the rolodex and an array of Booleans that
  ` will tell us if a query for the corresponding letter
  ` has been performed or not
      ARRAY STRING(1;asRolodex;26)
      ARRAY BOOLEAN(abQueryDone;26)
      For($vlElem;1;26)
         asRolodex{$vlElem}:=Char(64+$vlElem)
         abQueryDone{$vlElem}:=False
      End for

   :(Form event=On Clicked)
  ` When a click on the Tab control occurs, check whether the corresponding query
  ` has been performed or not
      If(Not(abQueryDone{asRolodex}))
  ` If not, redirect the next query(ies) toward a named selection
         SET QUERY DESTINATION(Into named selection;"temp")
  ` Perform the query
         QUERY([Phone Book];[Phone Book]Last name=asRolodex{asRolodex}+"@")
  ` Restore normal query mode
         SET QUERY DESTINATION(Into current selection)
  ` Use the records found
         USE NAMED SELECTION("temp")
         COPY NAMED SELECTION([Phone book];"Rolodex+asRolodex{asRolodex})
  ` Next time we choose that letter, we won't perform the query again
```

```
            abQueryDone{asRolodex}:=True
         Else
   ` Use the existing named selection for displaying the records corresponding to the chosen letter
            USE NAMED SELECTION("Rolodex"+asRolodex{asRolodex}
         End if

      :(Form event=On Unload)
   ` After the form disappeared from the screen
   ` Clear the named selections we created
         For($vlElem;1;26)
            If(abQueryDone{$vlElem})
               CLEAR NAMED SELECTION("Rolodex"+asRolodex{$vlElem})
            End if
         End for
   ` Clear the two arrays we no longer need
         CLEAR VARIABLE(asRolodex)
         CLEAR VARIABLE(abQueryDone)
   End case
```

## Example 2

The ***Unique values*** project method in this example allows you to verify the uniqueness of the values for any number of fields in a table. The current record can be an existing or a newly created record.

```
   //Unique values project method
   //Unique values ( Pointer ; Pointer { ; Pointer... } ) -> Boolean
   //Unique values ( ->Table ; ->Field { ; ->Field2... } ) -> Yes or No

   C_BOOLEAN($0)
   C_POINTER(${1})
   C_LONGINT($vlField;$vlNbFields;$vlFound;$vlCurrentRecord)
   $vlNbFields:=Count parameters-1
   $vlCurrentRecord:=Record number($1->)
   If($vlNbFields>0)
      If($vlCurrentRecord#-1)
         If($vlCurrentRecord<0)
   //The current record is an unsaved new record (record number is -3);
   //therefore we can stop the query as soon as at least one record is found
            SET QUERY LIMIT(1)
         Else
   //The current record is an existing record;
   //therefore we can stop the query as soon as at least two records are found
            SET QUERY LIMIT(2)
         End if
   //The query will return its result in $vlFound
   //without changing the current record nor the current selection
         SET QUERY DESTINATION(Into variable;$vlFound)
   //Make the query according to the number of fields that are specified
         Case of
            :($vlNbFields=1)
               QUERY($1->;$2->=$2->)
            :($vlNbFields=2)
               QUERY($1->;$2->=$2->;*)
               QUERY($1->;&;$3->=$3->)
            Else
               QUERY($1->;$2->=$2->;*)
               For($vlField;2;$vlNbFields-1)
                  QUERY($1->;&;${1+$vlField}->=${1+$vlField}->;*)
               End for
               QUERY($1->;&;${1+$vlNbFields}->=${1+$vlNbFields}->)
         End case
         SET QUERY DESTINATION(Into current selection) //Restore normal query mode
         SET QUERY LIMIT(0) //No longer limit queries
   //Process query result
         Case of
            :($vlFound=0)
               $0:=True //No duplicated values
            :($vlFound=1)
```

```
            If($vlCurrentRecord<0)
               $0:=False //Found an existing record with the same values as the unsaved new record
            Else
               $0:=True //No duplicated values; just found the very same record
            End if
         :($vlFound=2)
            $0:=False //Whatever the case is, the values are duplicated
      End case
   Else
      If(◊DebugOn) //Does not make sense; signal it if development version
         TRACE //WARNING! Unique values is called with NO current record
      End if
      $0:=False //Can't guarantee the result
   End if
Else
   If(◊DebugOn) //Does not make sense; signal it if development version
      TRACE //WARNING! Unique values is called with NO query condition
   End if
   $0:=False //Can't guarantee the result
End if
```

After this project method is implemented in your application, you can write:

```
 //...
If(Unique values(->[Contacts];->[Contacts]Company);->[Contacts]Last name;->[Contacts]First name)
 //Do appropriate actions for that record which has unique values
Else
   ALERT("There is already a Contact with this name for this Company.")
End if
 //...
```

# SET QUERY LIMIT

SET QUERY LIMIT ( limit )

| Parameter | Type | | Description |
|---|---|---|---|
| limit | Longint | ➡ | Number of records, or 0 for no limit |

## Description

**SET QUERY LIMIT** allows you to tell 4D to stop any subsequent query for the current process as soon as at least the number of records you pass in *limit* has been found.

For example, if you pass *limit* equal to 1, any subsequent query will stop browsing an index or the data file as soon as one record that matches the query conditions has been found.

To restore queries with no limit, call **SET QUERY LIMIT** again with *limit* equal to 0.

**Warning: SET QUERY LIMIT** affects all the subsequent queries made within the current process. REMEMBER to always counterbalance a call to **SET QUERY LIMIT***(limit)* (where *limit>0*) with a call to **SET QUERY LIMIT***(0)* in order to restore queries with no limit.

**SET QUERY LIMIT** changes the behavior of the query commands:

- **QUERY**
- **QUERY SELECTION**
- **QUERY BY EXAMPLE**
- **QUERY BY FORMULA**
- **QUERY BY SQL**
- **QUERY SELECTION BY FORMULA**
- **QUERY SELECTION WITH ARRAY**
- **QUERY WITH ARRAY**
- **QUERY BY ATTRIBUTE**
- **QUERY SELECTION BY ATTRIBUTE**

On the other hand, **SET QUERY LIMIT** does not affect the other commands that may change the current selection of a table, such as **ALL RECORDS**, **RELATE MANY**, and so on.

## Example 1

To perform a query corresponding to the request "...give me any ten customers whose gross sales are greater than $1 M...", you would write:

```
SET QUERY LIMIT(10)
QUERY([Customers];[Customers]Gross sales>1000000)
SET QUERY LIMIT(0)
```

## Example 2

See the second example for the **SET QUERY DESTINATION** command.

# Quick Report

- QR BLOB TO REPORT
- QR Count columns
- QR DELETE COLUMN
- QR DELETE OFFSCREEN AREA
- QR EXECUTE COMMAND
- QR Find column
- QR Get area property
- QR GET BORDERS
- QR Get command status
- QR GET DESTINATION
- QR Get document property
- QR Get drop column
- QR GET HEADER AND FOOTER
- QR Get HTML template
- QR GET INFO COLUMN
- QR Get info row
- QR Get report kind
- QR Get report table
- QR GET SELECTION
- QR GET SORTS
- QR Get text property
- QR GET TOTALS DATA
- QR GET TOTALS SPACING
- QR INSERT COLUMN
- QR MOVE COLUMN
- QR NEW AREA
- QR New offscreen area
- QR ON COMMAND
- QR REPORT
- QR REPORT TO BLOB
- QR RUN
- QR SET AREA PROPERTY
- QR SET BORDERS
- QR SET DESTINATION
- QR SET DOCUMENT PROPERTY
- QR SET HEADER AND FOOTER
- QR SET HTML TEMPLATE
- QR SET INFO COLUMN
- QR SET INFO ROW
- QR SET REPORT KIND
- QR SET REPORT TABLE
- QR SET SELECTION
- QR SET SORTS
- QR SET TEXT PROPERTY
- QR SET TOTALS DATA
- QR SET TOTALS SPACING

# QR BLOB TO REPORT

QR BLOB TO REPORT ( area ; blob )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| blob | BLOB | ➡ | BLOB that houses the report |

## Description

The **QR BLOB TO REPORT** command places the report contained in *blob* in the Quick Report area passed in *area*.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *blob* parameter, the error -9852 will be generated.

## Example 1

The following code allows you to display, in MyArea, a report file named “report.4qr” located next to the database structure. The report file does not have to be created with 4D version 2003; it can originate from previous versions:

```
C_BLOB($doc)
C_LONGINT(MyArea)
DOCUMENT TO BLOB("report.4qr";$doc)
QR BLOB TO REPORT(MyArea;$doc)
```

## Example 2

The following statement retrieves the Quick Report stored in Field4 and displays it in MyArea:

```
QR BLOB TO REPORT(MyArea;[Table 1]Field4)
```

# QR Count columns

QR Count columns ( area ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| Function result | Longint | ⇐ | Number of columns in area |

## Description

The **QR Count columns** command returns the number of columns present in the Quick Report *area*.

If you pass an invalid area number, the error -9850 will be generated.

## Example

The following code retrieves the column count and inserts a column to the right of the rightmost existing column:

```
$ColNb:=QR Count columns(MyArea)
QR INSERT COLUMN(MyArea;$ColNb+1;->[Table 1]Field2)
```

# QR DELETE COLUMN

QR DELETE COLUMN ( area ; colNumber )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| colNumber | Longint | ⇒ | Column number |

## Description

The **QR DELETE COLUMN** command deletes the column in *area* whose number was passed in *colNumber*. This command does not apply to cross-table reports.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *column* number, the error -9852 will be generated.

## Example

The following example makes sure the report is a list report and deletes the third column:

```
If(QR Get report kind(MyArea)=qr list report)
   QR DELETE COLUMN(MyArea;3)
End if
```

# QR DELETE OFFSCREEN AREA

QR DELETE OFFSCREEN AREA ( area )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area to delete |

## Description

The **QR DELETE OFFSCREEN AREA** command deletes in memory the Quick Report offscreen *area* whose reference was passed as parameter.

If you pass an invalid *area* number, the error -9850 will be generated.

# QR EXECUTE COMMAND

QR EXECUTE COMMAND ( area ; command )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| command | Longint | ⇒ | Menu command to be executed |

## Description

The **QR EXECUTE COMMAND** command executes the menu command or toolbar button whose reference was passed in *command*. The most common use for this command is to execute a command after the user selected that command and your code intercepted it through the **QR ON COMMAND** command.

In *command*, you can pass a value or one of the constants of the **QR Commands** constant theme:

| Constant | Type | Value |
| --- | --- | --- |
| qr cmd 4D View destination | Longint | 2503 |
| qr cmd add column | Longint | 2608 |
| qr cmd alt back color palette | Longint | 1004 |
| qr cmd automatic width | Longint | 2605 |
| qr cmd average | Longint | 507 |
| qr cmd back color palette | Longint | 1003 |
| qr cmd back colors toolbar | Longint | 2052 |
| qr cmd bold | Longint | 500 |
| qr cmd borders | Longint | 2609 |
| qr cmd center justified | Longint | 504 |
| qr cmd columns toolbar | Longint | 2054 |
| qr cmd count | Longint | 510 |
| qr cmd default justified | Longint | 512 |
| qr cmd delete column | Longint | 2601 |
| qr cmd disk file destination | Longint | 2501 |
| qr cmd edit column | Longint | 2603 |
| qr cmd font color palette | Longint | 1002 |
| qr cmd font dropdown | Longint | 1000 |
| qr cmd format | Longint | 2606 |
| qr cmd generate | Longint | 2008 |
| qr cmd graph destination | Longint | 2502 |
| qr cmd header and footer | Longint | 2005 |
| qr cmd hide column | Longint | 2602 |
| qr cmd hide line | Longint | 2607 |
| qr cmd HTML file destination | Longint | 2504 |
| qr cmd insert column | Longint | 2600 |
| qr cmd italic | Longint | 501 |
| qr cmd left justified | Longint | 503 |
| qr cmd max | Longint | 509 |
| qr cmd min | Longint | 508 |
| qr cmd move left | Longint | 3002 |
| qr cmd move right | Longint | 3003 |
| qr cmd new | Longint | 2000 |
| qr cmd open | Longint | 2001 |
| qr cmd operators toolbar | Longint | 2051 |
| qr cmd page setup | Longint | 2006 |
| qr cmd plain | Longint | 511 |
| qr cmd presentation | Longint | 2611 |
| qr cmd print preview | Longint | 2007 |
| qr cmd printer destination | Longint | 2500 |
| qr cmd repeated values | Longint | 2604 |
| qr cmd revert to save | Longint | 2004 |
| qr cmd right justified | Longint | 505 |
| qr cmd save | Longint | 2002 |
| qr cmd save as | Longint | 2003 |
| qr cmd standard deviation | Longint | 513 |
| qr cmd standard toolbar | Longint | 2053 |
| qr cmd style toolbar | Longint | 2050 |
| qr cmd sum | Longint | 506 |
| qr cmd totals spacing | Longint | 2610 |

| Constant | Type | Value |
| --- | --- | --- |
| qr cmd underline | Longint | 502 |

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *command* number, the error -9852 will be generated.

# QR Find column

QR Find column ( area ; expression ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| expression | String, Pointer | ➡ | Column object |
| Function result | Longint | ↩ | Number of the column |

## Description

The **QR Find column** command returns the number of the first column whose contents match the *expression* passed in parameter.

*expression* can either be a string or a pointer.

**QR Find column** returns -1 if nothing has been found.

If you pass an invalid area number, the error -9850 will be generated.

## Example

The following code retrieves the column number that holds the field [G.NQR Tests]Quarter and deletes that column:

```
$NumColumn:=QR Find column(MyArea;->[G.NQR Tests]Quarter)
```

or:

```
$NumColumn:=QR Find column(MyArea;"[G.NQR Tests]Quarter")
```

followed by:

```
If($NumColumn#-1)
   QR DELETE COLUMN(MyArea;$NumColumn)
End if
```

# QR Get area property

QR Get area property ( area ; property ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| property | Longint | ⇒ | Interface element designated |
| Function result | Longint | ⇐ | 1 = displayed, 0 = hidden |

## Description

The **QR Get area property** command returns 0 if the interface element (toolbar or menu bar) passed in *property* is not displayed; otherwise, it returns 1.

The menu bar and toolbars are numbered from 1 to 6 (top to bottom) and the value 7 is dedicated to the context menu.

You can use the constants from the **QR Area Properties** theme to designate the interface item:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr view color toolbar | Longint | 5 | Display status of the Color toolbar (Displayed=1, Hidden=0) |
| qr view column toolbar | Longint | 6 | Display status of the Column toolbar (Displayed=1, Hidden=0) |
| qr view contextual menus | Longint | 7 | Display status of the Contextual menu (Displayed=1, Hidden=0) |
| qr view menubar | Longint | 1 | Display status of the menu bar (Displayed=1, Hidden=0) |
| qr view operators toolbar | Longint | 4 | Display status of the Operators toolbar (Displayed=1, Hidden=0) |
| qr view standard toolbar | Longint | 2 | Display status of the Standard toolbar (Displayed=1, Hidden=0) |
| qr view style toolbar | Longint | 3 | Display status of the Style toolbar (Displayed=1, Hidden=0) |

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *property* parameter, the error -9852 will be generated.

# QR GET BORDERS

QR GET BORDERS ( area ; column ; row ; border ; line {; color} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| column | Longint | ⇒ | Column number |
| row | Longint | ⇒ | Row number |
| border | Longint | ⇒ | Border value |
| line | Longint | ⇐ | Line thickness |
| color | Longint | ⇐ | Border color |

## Description

The **QR GET BORDERS** command allows you to retrieve the border style for a border of a given cell.

*area* is the reference of the Quick Report area.

*column* is the column number of the cell.

*row* designates the row number of the cell. You can either:

- pass a positive integer value to designate the corresponding subtotal (break) level that is affected.
- pass one of the following constants of the **QR Rows for Properties** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr detail | Longint | -2 | Detail area of report |
| qr grand total | Longint | -3 | Grand total area |
| qr title | Longint | -1 | Title of report |

*border* is the value that indicates which cell border is affected. Pass one of the constants from the **QR Borders** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr bottom border | Longint | 8 | Bottom border |
| qr inside horizontal border | Longint | 32 | Inside horizontal border |
| qr inside vertical border | Longint | 16 | Inside vertical border |
| qr left border | Longint | 1 | Left border |
| qr right border | Longint | 4 | Right border |
| qr top border | Longint | 2 | Top border |

**Note:** Unlike the command **QR SET BORDERS**, **QR GET BORDERS** does not accept a cumulative value. You must test all the parameters separately to have an overall view of the cell border.

*line* is the thickness of the line:

- 0 indicates no line
- 1 indicates a thickness of 1/4 point
- 2 indicates a thickness of 1/2 point
- 3 indicates a thickness of 1 point
- 4 indicates a thickness of 2 points.

*color* is the color of the line; it returns the value of the color applied to the line segment.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *column* number, the error -9852 will be generated.
If you pass an invalid *row* number, the error -9853 will be generated.
If you pass an invalid *border* parameter, the error -9854 will be generated.

# QR Get command status

QR Get command status ( area ; command {; value} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| command | Longint | ⇒ | Command number |
| value | Longint, Text | ⇐ | Value for the selected sub-item |
| Function result | Longint | ⇐ | Command status |

## Description

The **QR Get command status** command returns 0 if the *command* is disabled or 1 if it is enabled.

*value* returns the value of the selected sub-item, if any. For example, if the command that was selected is the **Font** menu (1000) and the font selected was “Arial”, *value* would return “Arial”, or if the command that was selected is a color menu (1002, 1003 or 1004), *value* would return the color number.

You can use the command in two types of contexts:

- As a simple statement to determine whether a command is enabled or disabled.
- In the method installed by **QR ON COMMAND**, to allow you to know which sub-item was selected. In that method, *$1* is the reference of the area and *$2* is the number of the command.

In *command*, you can pass a value or one of the constants of the **QR Commands** constant theme.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *command* number, the error -9852 will be generated.

# QR GET DESTINATION

QR GET DESTINATION ( area ; type {; specifics} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| type | Longint | ⇐ | Type of the report |
| specifics | String, Variable | ⇐ | Specifics linked to the output type |

## Description

The **QR GET DESTINATION** command retrieves the output *type* of the report for the area whose reference was passed in *area*.

You can compare the value of the *type* parameter with the constants of the **QR Output Destination** theme.

The following table describes the values that can be retrieved in both *type* and *specifics* parameters:

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_qr 4D Chart area | Longint | 4 | *** Obsolete constant *** |
| qr 4D View area | Longint | 3 | *specifics*: N.A. |
| qr HTML file | Longint | 5 | *specifics*: Pathname to the file. |
| qr printer | Longint | 1 | *specifics*: "*" to remove the print dialog boxes |
| qr text file | Longint | 2 | *specifics*: Pathname to the file. |

If you pass an invalid *area* number, the error -9850 will be generated.

# QR Get document property

QR Get document property ( area ; property ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| property | Longint | ⇒ | 1 = Print Dialog, 2 = Document unit |
| Function result | Longint | ⇐ | Value for the property |

## Description

The **QR Get document property** command retrieves the display status for the print dialog box or the unit used for the document that are present in *area*.

In *property*, you can use the following constants, located in the **QR Document Properties** constant theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr printing dialog | Longint | 1 | Display of the print dialog box:<br>• If value = 0, the print dialog is not displayed prior to printing.<br>• If value = 1, the print dialog is displayed prior to printing (default value). |
| qr unit | Longint | 2 | Document unit:<br>• If value = 0, the document unit is points.<br>• If value = 1, the document unit is centimeters.<br>• If value = 2, the document unit is inches. |

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *property* value, the error -9852 will be generated.

# QR Get drop column

QR Get drop column ( area ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| Function result | Longint | ⇐ | Drop value |

## Description

The **QR Get drop column** command returns a value depending on where the drop was performed:

- if the value is negative, it indicates a column number (i.e., -3 if the the drop was performed on column number 3)
- if the value is positive, it indicates that the drop was performed on a separator preceding the column (i.e., 3 if the drop was performed after column 2). Keep in mind that the drop does not have to take place before an existing column.

If you pass an invalid *area* number, the error -9850 will be generated.

# QR GET HEADER AND FOOTER

QR GET HEADER AND FOOTER ( area ; selector ; leftTitle ; centerTitle ; rightTitle ; height {; picture {; pictAlignment}} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| selector | Longint | ➡ | 1 = Header, 2 = Footer |
| leftTitle | String | ⬅ | Text displayed on the left side |
| centerTitle | String | ⬅ | Text displayed in the middle |
| rightTitle | String | ⬅ | Text displayed on the right side |
| height | Longint | ⬅ | Header or footer height |
| picture | Picture | ⬅ | Picture to display |
| pictAlignment | Longint | ⬅ | Alignment attribute for the picture |

## Description

The **QR GET HEADER AND FOOTER** command retrieves the contents and size of the header or footer.

*selector* allows you to select the header or the footer:

- if *selector* equals 1, the header information will be retrieved;
- if *selector* equals 2, the footer information will be retrieved.

*leftTitle*, *centerTitle* and *rightTitle* returns the values for, respectively, the left, center and right header/footer.

*height* returns the height of the header/footer, expressed in the unit selected for the report.

*picture* returns a picture that is displayed in the header or footer.

*pictAlignment* is the alignment attribute for the picture displayed in the header/footer.

- If *pictAlignment* returns 1, the picture is aligned to the left.
- If *pictAlignment* returns 2, the picture is centered.
- If *pictAlignment* returns 3, the picture is aligned to the right.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *selector* value, the error -9852 will be generated.

## Example

The following code retrieves the values of the header titles as well as the header size and displays them in alerts:

```
QR GET HEADER AND FOOTER(MyArea;1;$LeftText;$CenterText;$RightText;$height)
Case of
   :($LeftText #"")
      ALERT("The left title is "+Char(34)+$LeftText+Char(34))
   :($CenterText #"")
      ALERT("The center title is "+Char(34)+$CenterText+Char(34))
   :($RightText #"")
      ALERT("The right title is "+Char(34)+$RightText+Char(34))
   Else
      ALERT("No header title in this report.")
End case
ALERT("The height of the header is "+String($height))
```

# QR Get HTML template

QR Get HTML template ( area ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| Function result | Text | ⮌ | HTML code used as template |

## Description

The **QR Get HTML template** command returns the HTML template currently used for the Quick Report *area*. The returned value is a text value and includes all the contents of the HTML template.

If no specific template was defined, the template that is returned is the default template. Please note that no template will be returned if the output was not set to HTML file, either manually or programmatically.

If you pass an invalid area number, the error -9850 will be generated.

# QR GET INFO COLUMN

QR GET INFO COLUMN ( area ; colNum ; title ; object ; hide ; size ; repeatedValue ; displayFormat {; resultVar} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| colNum | Longint | ➡ | Column number |
| title | Text | ⬅ | Column title |
| object | Text | ⬅ | Object assigned for that column |
| hide | Longint | ⬅ | 0 = displayed, 1 = hidden |
| size | Longint | ⬅ | Column size |
| repeatedValue | Longint | ⬅ | 0 = not repeated, 1 = repeated |
| displayFormat | Text | ⬅ | Display format for the data |
| resultVar | Text | ⬅ | Name of the formula variable |

## Description

### List mode

The **QR GET INFO COLUMN** command retrieves the parameters of an existing column.

*area* is the reference of the Quick Report area.

*colNum* is the number of the column to modify.

*title* returns the title that will be displayed in the header of the column.

*object* returns the field name or the formula assigned to the column.

**Note:** The command does not take into account any virtual structure defined by means of the **SET TABLE TITLES** and **SET FIELD TITLES** commands. The actual name of the field is returned in the *object* parameter.

*hide* returns whether the column is displayed or hidden:

- if *hide* equals 1, the column is hidden;
- if *hide* equals 0, the column is displayed.

*size* returns the size of the column in pixels. If the value returned is negative, the size of the column is automatic.

*repeatedValue* returns the status for data repetition. For example, if the value for a field or variable does not change from one record to the other, it may or may not be repeated when they do not change:

- if *repeatedValue* equals 0, values are not repeated,
- if *repeatedValue* equals 1, values are repeated.

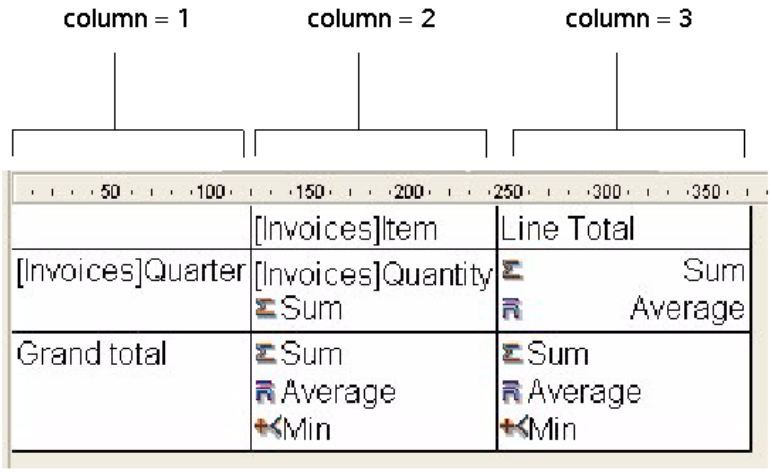
*format* returns the display format. Display formats are the 4D formats compatible with the data displayed.

When passed, the optional *resultVar* parameter returns the name of the variable automatically assigned by the Quick Report editor to the formula column (if any): "C1" for the first formula column, "C2" for the second, and so on. 4D uses this variable to store the results from the last execution of the column's formula when generating the report.

### Cross-table mode

The **QR GET INFO COLUMN** command retrieves the same parameters but the reference of the areas to which it applies is different and varies depending on the parameter you want to set. First of all, the *title*, *hide*, and *repeatedValue* parameters are meaningless when this command is used in cross-table mode. The value to use for *colNum* varies depending on whether you want to retrieve the column size or the data source and display format.

- Column size
  This is a “visual” attribute, therefore columns are numbered from left to right, as depicted below:

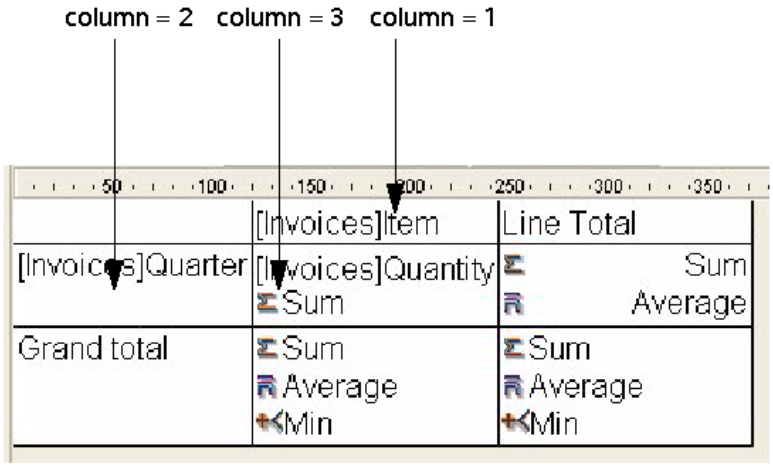



The following statement sets the size to automatic for all the columns in a cross-table report and leaves other elements unchanged:

```
For($i;1;3)
   QR GET INFO COLUMN(qr_area;$i;$title;$obj;$hide;$size;$rep;$format)
   QR SET INFO COLUMN(qr_area;$i;$title;$obj;$hide;0;$rep;$format)
End for
```

You will notice that since you want to alter only the column size, you have to use **QR GET INFO COLUMN** to retrieve the column properties and pass them to **QR SET INFO COLUMN** to leave it unchanged, except for the column size.

- Data source (object) and display format
  In this case, the numbering of columns operates as depicted below:




If you pass an invalid area number, the error -9850 will be generated.
If you pass an invalid ColNum value, the error -9852 will be generated.

## Example

You have designed the following report:

| | [People]LastName | [People]FirstName | C1 | [People]City | [People]Salary |
|---|---|---|---|---|---|
| Title | LastName | FirstName | Age | City | Salary |
| Detail | | | | | |
| Grand total | | | | | |

You can write:

```
C_TEXT($vTitle;$vObject;$vDisplayFormat;$vResultVar)
C_LONGINT($area;$vHide;$vSize;$vRepeatedValue)
QR GET INFO COLUMN($area;3;$vTitle;$vObject;$vHide;$vSize;$vRepeatedValue;$vDisplayFormat;$vResultVar)
 //$vTitle = "Age"
 //$vObject = "[People]Birthdate-Current date"
 //$vHide = 0
 //$vSize = 57
 //$vRepeatedValue = 1
```

```
    //$vDisplayFormat = ""
    //$vResultVar = "C1"
```

# QR Get info row

QR Get info row ( area ; row ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area created |
| row | Longint | ⇒ | Row designator |
| Function result | Longint | ⇐ | 0 = displayed, 1 = hidden |

## Description

The **QR Get info row** command retrieves the display status of the row whose reference was passed in *row*.

*row* designates which row is affected by the command. You can pass either:

- a positive integer value to designate the corresponding subtotal (break) level,
- one of the following constants from the **QR Rows for Properties** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr detail | Longint | -2 | Detail area of report |
| qr grand total | Longint | -3 | Grand total area |
| qr title | Longint | -1 | Title of report |

The value returned by **QR Get info row** indicates whether the contents of the row are displayed or hidden. If it equals 1, the contents of the row are hidden; if it equals 0, the contents of the row are displayed.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *row* value, the error -9852 will be generated.

# QR Get report kind

QR Get report kind ( area ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| Function result | Longint | ⇐ | Type of the report |

## Description

The **QR Get report kind** command retrieves the report type for the area whose reference was passed in *area*.

- If the command returns 1, the report type is list.
- If the command returns 2, the report type is cross-table.

You can also compare the function result with the constants of the **QR Report Types** theme:

| Constant | Type | Value |
|---|---|---|
| qr cross report | Longint | 2 |
| qr list report | Longint | 1 |

If you pass an invalid *area* number, the error -9850 will be generated.

# QR Get report table

QR Get report table ( area ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| Function result | Longint | ⇐ | Table number |

## Description

The **QR Get report table** command returns the current table number for the report area whose reference was passed in *area*.

If you pass an invalid *area* number, the error -9850 will be generated.

# QR GET SELECTION

QR GET SELECTION ( area ; left ; top {; right {; bottom}} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| left | Longint | ⇐ | Left boundary |
| top | Longint | ⇐ | Top boundary |
| right | Longint | ⇐ | Right boundary |
| bottom | Longint | ⇐ | Bottom boundary |

## Description

The **QR GET SELECTION** command returns the coordinates of the cell that is selected.

*left* returns the number of the column that is the left boundary of the selection. If *left* equals 0, the entire row is selected.

*top* returns the number of the row that is the top boundary of the selection. If *top* equals 0, the entire column is selected.

**Note:** If both *left* and *top* equal 0, the entire area is highlighted.

*right* is the number of the column that is the right boundary of the selection.

*bottom* is the number of the row that is the top boundary of the selection.

**Note:** If there is no selection, *left*, *top*, *right* and *bottom* are set to -1.

If you pass an invalid *area* number, the error -9850 will be generated.

# QR GET SORTS

QR GET SORTS ( area ; aColumns ; aOrders )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇨ | Reference of the area |
| aColumns | Real array | ⇦ | Sorted columns |
| aOrders | Real array | ⇦ | Sort orders |

## Description

The **QR GET SORTS** command populates two arrays:

- *aColumns*
  This array includes all the columns that have a sort order.
- *aOrders*
  Each element of this array contains the sort orders for the matching column.
  - If *aOrders{$i}* equals 1, the sort order is ascending.
  - If *aOrders{$i}* equals -1, the sort order is descending.

### Cross-table mode

In the case of cross-table mode, the resulting arrays cannot have more than two elements since sorts can only be performed on columns (1) and rows (2). (Values for *aColumns*).

If you pass an invalid *area* number, the error -9850 will be generated.

# QR Get text property

QR Get text property ( area ; colNum ; rowNum ; property ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| colNum | Longint | ⇒ | Column number |
| rowNum | Longint | ⇒ | Row number |
| property | Longint | ⇒ | Property number |
| Function result | Longint, String | ⇐ | Value for the selected property |

## Description

The **QR Get text property** command returns the property value of the text attributes for the cell determined by *colNum* and *RowNum*.

*area* is the reference of the Quick Report area.

*colNum* is the number of the cell column.

*rowNum* is the reference of the cell row. You can either pass:

- a positive value designating the corresponding subtotal (break) level,
- one of the constants of the **QR Rows for Properties** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr detail | Longint | -2 | Detail area of report |
| qr footer | Longint | -5 | Page footer |
| qr grand total | Longint | -3 | Grand total area |
| qr header | Longint | -4 | Page header |
| qr title | Longint | -1 | Title of report |

**Note:** When passing -4 or -5 as *rowNum*, you still need to pass a column number in *colNum*, even if it is not used.

**Note:** In cross-table mode, the principle is similar except for the row values, which are always positive.

*property* is the value of the text attribute to get. You can use the constants of the **QR Text Properties** theme, and the following values can be returned:

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_qr font | Longint | 1 | Obsolete since 4D v14R3 (use qr font name) |
| qr alternate background color | Longint | 9 | Alternate background color number |
| qr background color | Longint | 8 | Background color number |
| qr bold | Longint | 3 | Bold style attribute (0 or 1) |
| qr font name | Longint | 10 | Name of font as returned for example by the **FONT LIST** command |
| qr font size | Longint | 2 | Font size expressed in points (9 to 255) |
| qr italic | Longint | 4 | Italic style attribute (0 or 1) |
| qr justification | Longint | 7 | Justification attribute (0 for default, 1 for left, 2 for center or 3 for right) |
| qr text color | Longint | 6 | Color number attribute (Longint) |
| qr underline | Longint | 5 | Underline style attribute (0 or 1) |

If you pass an invalid *area* number, the error -9850 will be generated.

If you pass an invalid *colNum* number, the error -9852 will be generated.

If you pass an invalid *rowNum* number, the error -9853 will be generated.

If you pass an invalid *property* number, the error -9854 will be generated.

# QR GET TOTALS DATA

QR GET TOTALS DATA ( area ; colNum ; breakNum ; operator ; text )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| colNum | Longint | ⇒ | Column number |
| breakNum | Longint | ⇒ | Break number |
| operator | Longint | ⇐ | Operator value for the cell |
| text | String | ⇐ | Contents of the cell |

## Description

### List Mode

The **QR GET TOTALS DATA** command retrieves the details of a specific break.

*area* is the reference of the Quick Report area.

*colNum* is the number of the column whose data will be retrieved.

*breakNum* is the number of the break whose data will be retrieved (subtotal or grand total). For a subtotal row, *breakNum* corresponds to the row number. For a grand total, *breakNum* is -3 (you can also use the *qr grand total* constant from the **QR Rows for Properties** theme).

*operator* returns the sum of all the operators present in the cell. You can use the constants of the **QR Operators** theme to process the returned value:

| Constant | Type | Value |
|---|---|---|
| qr average | Longint | 2 |
| qr count | Longint | 16 |
| qr max | Longint | 8 |
| qr min | Longint | 4 |
| qr standard deviation | Longint | 32 |
| qr sum | Longint | 1 |

If the value returned is 0, there is no operator.

*text* returns the text present in the cell.

**Note:** *operator* and *text* are mutually exclusive, so you either have a result returned through *operator* or through *text*.

### Cross-table Mode

The **QR GET TOTALS DATA** command retrieves the details of a specific cell.

*area* is the reference of the Quick Report area.

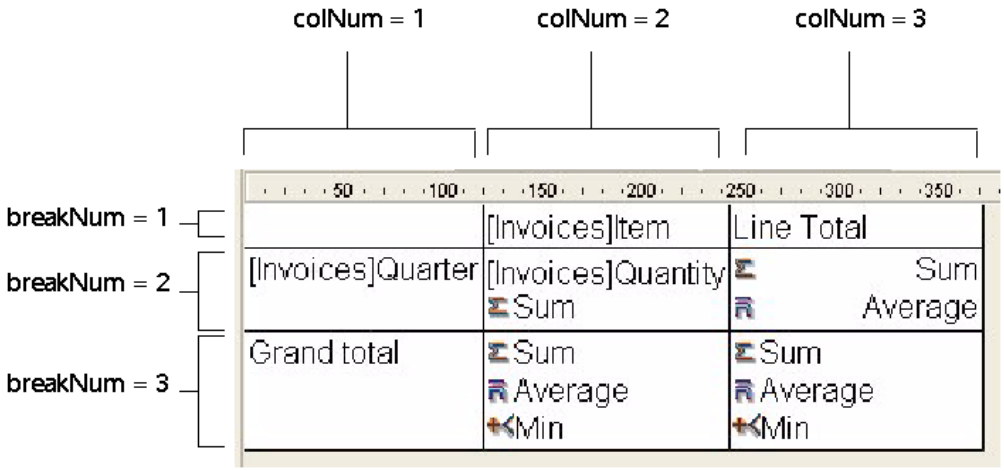
*colNum* is the column number of the cell whose data is going to be retrieved.

*breakNum* is the row number of the cell whose data is going to be retrieved.

*operator* returns the sum of all the operators present in the cell. You can use the constants of the **QR Operators** theme to process the returned value (see above).

*text* returns the text in the cell.

Here is a depiction of how the parameters *colNum* and *breakNum* have to be combined in cross-table mode:




If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *colNum* number, the error -9852 will be generated.
If you pass an invalid *breakNum* number, the error -9853 will be generated.

# QR GET TOTALS SPACING

QR GET TOTALS SPACING ( area ; subtotal ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| subtotal | Longint | ➡ | Subtotal number |
| value | Longint | ⬅ | 0=no space, 32000=inserts a page break, >0=spacing added below the break level, <0=proportional increase |

## Description

The **QR GET TOTALS SPACING** command retrieves a space below a subtotal row. It applies only to the list mode.

*area* is the reference of the Quick Report area.

*subtotal* is the subtotal level (or break level) that will be affected. *subtotal* is a value between 1 and the number of the subtotal/sort.

*value* defines the value of the spacing:

- If *value* is 0, no space is added.
- If *value* is 32000, a page break is inserted.
- If *value* is a positive value, it expresses the spacing value in pixels.
- If *value* is a negative value, it expresses the spacing as a percentage of the subtotal row. For example, -100 will set a space of 100% below the subtotal row.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *subtotal*, the error -9852 will be generated.

# QR INSERT COLUMN

QR INSERT COLUMN ( area ; colNumber ; object )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| colNumber | Longint | ➡ | Column number |
| object | Field, Variable, Pointer | ➡ | Object to be inserted in the column |

## Description

The **QR INSERT COLUMN** command inserts or creates a column at the specified position. Columns located to the right of that position will be shifted accordingly.

*position* is the number of the column, established from left to right.

The default title for the column will be the value passed in *object*.

If you pass an invalid *area* number, the error -9850 will be generated.

**Note:** This command cannot be used with a cross-table report.

## Example

The following statement inserts (or creates) a first column in a Quick Report area, inserts “Field1” as column title (default behavior) and populates the contents of the body with values from Field1.

```
QR INSERT COLUMN(MyArea;1;->[Table 1]Field1)
```

# QR MOVE COLUMN

QR MOVE COLUMN ( area ; column ; newPos )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| column | Longint | ➡ | Column number |
| newPos | Longint | ➡ | New position for column |

## Description

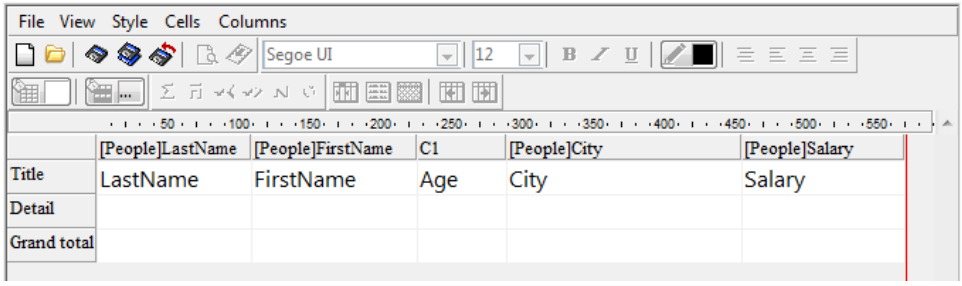
The **QR MOVE COLUMN** command moves the column currently found at the *column* position to the *newPos* position.

Both the *column* and *newPos* parameters must be valid column numbers (between 1 and the total number of columns in the report); otherwise, the error -9852 is returned.

**Note:** This command can be used with list reports only.

## Example

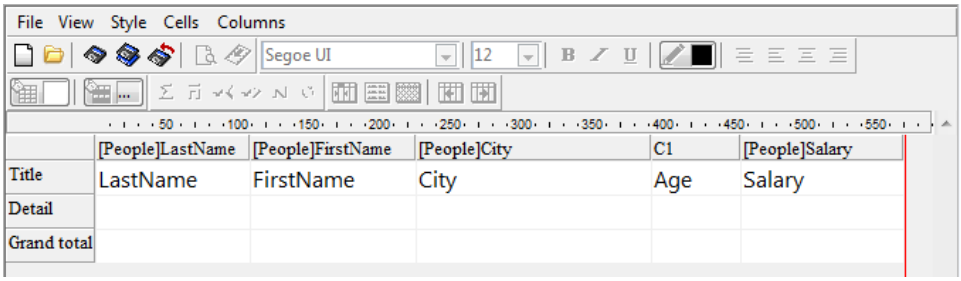
You have designed the following report:




If you execute:

```
QR MOVE COLUMN(area;3;4)
```

The result is:



# QR NEW AREA

QR NEW AREA ( ptr )

| Parameter | Type | | Description |
|---|---|---|---|
| ptr | Pointer | ⇨ | Pointer to a variable |

## Description

The **QR NEW AREA** command creates a new Quick Report area and stores its reference number in the longint variable referenced by the *ptr* pointer.

# QR New offscreen area

QR New offscreen area -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ⮌ | Reference of the area created |

## Description

The **QR New offscreen area** command creates a new Quick Report offscreen area and returns its reference.

# QR ON COMMAND

QR ON COMMAND ( area ; methodName )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| methodName | String | ⇒ | Name of the replacement method |

## Description

The **QR ON COMMAND** command executes the 4D method passed in *methodName* when a Quick Report command is invoked by the user, by the selection of a menu command or by a click on a button.

If *area* equals zero, *methodName* will apply to each Quick Report area until the database is closed or until the following call to the command is made: **QR ON COMMAND**(0;"").

*methodName* receives two parameters:

- *$1* is the reference of the area (Longint).
- *$2* is the command number of the command that was selected (Longint). You can compare this value with the constants of the **QR Commands** theme.

**Note:** When planning on compiling the database, it is necessary to declare both *$1* and *$2* as Longints, even if you do not use them.

If you want the initial command to be executed, you need to include the following in the called method: **QR EXECUTE COMMAND**($1;$2).

If you pass an invalid *area* number, the error -9850 will be generated.

# QR REPORT

QR REPORT ( {aTable ;} document {; hierarchical {; wizard {; search {; methodName {; *}}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to use for the report, or Default table if omitted |
| document | String | ⇒ | Quick Report document to load |
| hierarchical | Boolean | ⇒ | True = Display related Many tables, False or omitted = Do not display (default) |
| wizard | Boolean | ⇒ | True = Display the wizard button, False or omitted = Do not display (default) |
| search | Boolean | ⇒ | True = Display the search tools and master table choice, False or omitted = Do not display (default) |
| methodName | String | ⇒ | Name of method to call |
| * | Operator | ⇒ | Deletion of printing dialog boxes |

## Description

**QR REPORT** prints a report for *aTable*, created with the Quick Report editor. This editor allows users to create their own reports. For more information about creating reports with the Quick Report editor, refer to the **Quick reports** or **Quick reports (64-bit)** sections of the 4D *Design Reference* manual.

**Notes:**

- The editor does not appear if the *table* has been declared “Invisible.”
- When the editor is called using the **QR REPORT** command, relations between tables keep their manual status, where applicable. This allows the developer to manage this status himself using the **SET AUTOMATIC RELATIONS** and **SET FIELD RELATION** command. In 32-bit versions, the **All relations in automatic** option that is used to modify the automatic/manual status of the relations is hidden.
- The editor is called in an external window and it is not possible to use the **QR ON COMMAND** command in this context. However, you can use the *methodName* parameter to execute custom code when an interface command is activated (see below).

The *document* parameter is a report document that was created with the Quick Report editor and saved on disk. The document stores the specifications of the report, not the records to be printed.
If an empty string ("") is specified for *document*, **QR REPORT** displays an Open File dialog box and the user can select the report to print.
If the *document* parameter specifies a document that does not exist (for example, pass **Char(1)** in *document*), the Quick Report editor is displayed.

32-bit versions only:

- The *hierarchical* parameter defines whether the related Many tables are displayed in the field selection list. By default, this value is set to 0 (no display for related Many tables).
- The *wizard* parameter indicates whether the Open Wizard button is going to be displayed in the Quick Report editor, therefore either allowing or disallowing access to the wizard. By default, this value is set to False (no access to the wizard).
- The *search* parameter indicates whether the New Query button and the Master table drop-down menu are going to be displayed in the Quick Report editor, therefore either allowing or disallowing modification of the current table and current master table. By default, this value is set to False (no access to the search tools and master table).
- *The methodName* parameter designates a 4D project method that will be executed each time a command of the Quick Report editor is called by selecting a menu item or clicking on a button. Using this parameter is equivalent to using **QR ON COMMAND** in the context of the Quick Report editor window (**QR ON COMMAND** only works within the context of an included area). More particularly, you can use this new parameter to change the character set used by the quick report.
  The *methodName* method receives two parameters:
  - *$1* contains the area reference (longint).
  - *$2* contains the number of the command selected (longint). You can compare this value with the constants of the **QR Commands** theme.

  **Note:** If you want to compile your database using the Compiler, you must declare the $1 et $2 parameters explicitly as longints, even if you do not use them.

  If you want to execute the initial command chosen by the user, use the following statement in the *methodName* method:

```
QR EXECUTE COMMAND($1;$2)
```

If the *methodName* parameter is an empty string ("") or is omitted, no method is called and the standard operation of **QR REPORT** is applied.

After a report is selected, the dialog boxes for printing are displayed, unless the * parameter is specified. If this parameter is specified, these dialog boxes are not displayed. The report is then printed.

If the Quick Report editor is not involved, the OK variable is set to 1 if a report is printed; otherwise, it is set to 0 (zero) (i.e., if the user clicked **Cancel** in the printing dialog boxes).

**4D Server:** This command can be executed on 4D Server within the framework of a stored procedure. In this context:

- Make sure that no dialog box appears on the server machine (except for a specific requirement). To do this, it is necessary to call the command with the * parameter.
- The syntax which makes the Quick Report editor appear does not work with 4D Server; in this case, the system variable OK is set to 0.
- In the case of a problem concerning the printer (out of paper, printer disconnected, etc.), no error message is generated.

## Example 1

The following example lets the user query the [People] table, and then automatically prints the report “Detailed Listing”:

```
QUERY([People])
If(OK=1)
   QR REPORT([People];"Detailed Listing";False;False;False;*)
End if
```

## Example 2

The following example lets the user query the [People] table, and then lets the user choose which report to print:

```
QUERY([People])
If(OK=1)
   QR REPORT([People];"";False;False;False)
End if
```

## Example 3

The following example lets the user query the [People] table, and then displays the Quick Report editor so the user can design, save, load and print any reports with or without the wizard:

```
QUERY([People])
If(OK=1)
   QR REPORT([People];Char(1);False;True)
End if
```

## Example 4

Refer to the example of the **SET FIELD RELATION** command.

## Example 5

You want to convert the character set used in a quick report called using **QR REPORT** into Mac Roman:

```
QR REPORT([MyTable];Char(1);False;False;False;"myCallbackMeth")
```

The ***myCallbackMeth*** method converts the report when it is generated:

```
C_LONGINT($1;$2)
If($2=gr cmd generate) //if we generated a report
   C_BLOB($myblob)
   C_TEXT($path;$text)
   C_LONGINT($type)
   QR EXECUTE COMMAND($1;$2) //execution of command
   QR GET DESTINATION($1;$type;$path) //retrieval of destination
   If(($type=gr HTML file)|($type=gr text file))
      DOCUMENT TO BLOB($path;$myblob)
 //conversion to text using UTF-8
      $text:=Convert to text($myblob;"UTF-8")
 //use of MacRoman set
      CONVERT FROM TEXT($text;"MacRoman";$myblob)
 //Return of converted report
      BLOB TO DOCUMENT($path;$myblob)
   End if
Else //otherwise, execution of the command
   QR EXECUTE COMMAND($1;$2)
End if
```

# QR REPORT TO BLOB

QR REPORT TO BLOB ( area ; blob )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇨ | Reference of the area |
| blob | BLOB | ⇦ | BLOB to house the Quick Report |

## Description

The **QR REPORT TO BLOB** command places the report whose reference was passed in *area* in a BLOB (variable or field).
If you pass an invalid *area* number, the error -9850 will be generated.

## Example

The following statement assigns the Quick Report stored in MyArea into a BLOB Field.

```
QR REPORT TO BLOB(MyArea;[Table 1]Field4)
```

# QR RUN

QR RUN ( area )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area to execute |

## Description

The **QR RUN** command executes the report *area* whose reference was passed as parameter with the Quick Report current settings, including the output type. You can use the **QR SET DESTINATION** command to modify the output type.

The report is executed on the table to which the area belongs. When *area* designates an offscreen area, it is necessary to specify the table to be used via the **QR SET REPORT TABLE** command.

If you pass an invalid *area* number, the error -9850 will be generated.

**4D Server:** This command can be executed on 4D Server as part of a stored procedure. In this context, make sure that no dialog box appears on the server machine (except for specific requirements). To do this, you need to call the **QR SET DESTINATION** command with the "*" parameter. In case of a printer problem (out of paper, printer disconnected, etc.), no error message is generated.

# QR SET AREA PROPERTY

QR SET AREA PROPERTY ( area ; property ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| property | Longint | ⇒ | Interface element designated |
| value | Longint | ⇒ | 1 = displayed, 0 = hidden |

## Description

The **QR SET AREA PROPERTY** command shows or hides the interface element (toolbar or menu bar) whose reference is passed in *property*.

The menu bar and toolbars are numbered from 1 to 6 (top to bottom) and the value 7 is dedicated to the context menu. You can use the constants from the **QR Area Properties** theme to designate the interface item:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr view color toolbar | Longint | 5 | Display status of the Color toolbar (Displayed=1, Hidden=0) |
| qr view column toolbar | Longint | 6 | Display status of the Column toolbar (Displayed=1, Hidden=0) |
| qr view contextual menus | Longint | 7 | Display status of the Contextual menu (Displayed=1, Hidden=0) |
| qr view menubar | Longint | 1 | Display status of the menu bar (Displayed=1, Hidden=0) |
| qr view operators toolbar | Longint | 4 | Display status of the Operators toolbar (Displayed=1, Hidden=0) |
| qr view standard toolbar | Longint | 2 | Display status of the Standard toolbar (Displayed=1, Hidden=0) |
| qr view style toolbar | Longint | 3 | Display status of the Style toolbar (Displayed=1, Hidden=0) |

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *property* parameter, the error -9852 will be generated.

# QR SET BORDERS

QR SET BORDERS ( area ; column ; row ; border ; line {; color} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| column | Longint | ➡ | Column number |
| row | Longint | ➡ | Row number |
| border | Longint | ➡ | Border composite value |
| line | Longint | ➡ | Line thickness |
| color | Longint | ➡ | Border color |

## Description

The **QR SET BORDERS** command sets the border style for a given cell.

*area* is the reference of the Quick Report area.

*column* is the column number of the cell.

*row* is the row number of the cell. You can pass either:

- a positive integer value to designate the corresponding subtotal (break) level,
- one of the following constants located in the **QR Rows for Properties** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr detail | Longint | -2 | Detail area of report |
| qr grand total | Longint | -3 | Grand total area |
| qr title | Longint | -1 | Title of report |

*border* is a composite value that indicates which borders of the cell are to be affected. Pass one of the constants from the **QR Borders** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr bottom border | Longint | 8 | Bottom border |
| qr inside horizontal border | Longint | 32 | Inside horizontal border |
| qr inside vertical border | Longint | 16 | Inside vertical border |
| qr left border | Longint | 1 | Left border |
| qr right border | Longint | 4 | Right border |
| qr top border | Longint | 2 | Top border |

*border* can contain an accumulation of several values in order to designate several borders simultaneously. For example, a value of 5 passed in *border* would affect the right and left borders.
*line* is the thickness of the line:

- 0 indicates no line
- 1 indicates a thickness of 1/4 point
- 2 indicates a thickness of 1/2 point
- 3 indicates a thickness of 1 point
- 4 indicates a thickness of 2 points

*color* is the color of the line:

- If *color* is a positive value, it indicates a specific color.
- If *color* equals 0, the color is black.
- If *color* equals -1, no changes are to be made.

**Note:** The default color is black.

If you pass an invalid area number, the error -9850 will be generated.
If you pass an invalid column number, the error -9852 will be generated.
If you pass an invalid row number, the error -9853 will be generated.
If you pass an invalid border parameter, the error -9854 will be generated.
If you pass an invalid line parameter, the error -9855 will be generated.

# QR SET DESTINATION

QR SET DESTINATION ( area ; type {; specifics} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| type | Longint | ⇒ | Type of the report |
| specifics | String, Variable | ⇒ | Specifics linked to the output type |

## Description

The **QR SET DESTINATION** command sets the output *type* of the report for the area whose reference was passed in *area*.

In the *type* parameter, you can pass one of the constants of the **QR Output Destination** theme. The contents of the *specifics* parameter depends on the value of *type*. The following table describes the values that can be passed in both *type* and *specifics* parameters:

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_qr 4D Chart area | Longint | 4 | *** Obsolete constant *** |
| qr 4D View area | Longint | 3 | *specifics*: N.A. |
| qr HTML file | Longint | 5 | *specifics*: Pathname to the file. |
| qr printer | Longint | 1 | *specifics*: "*" to remove the print dialog boxes |
| qr text file | Longint | 2 | *specifics*: Pathname to the file. |

*qr printer* (1): If you pass a string containing a star ("*") in the *specifics* parameter, no dialog box will be displayed during printing and the current print settings will be used automatically. This setting is necessary when you want to print the report on the server.

*qr text file* (2): If you pass an empty string in the *specifics* parameter, a Save file dialog is displayed; otherwise the file is saved at the location indicated by the path.
The default field delimiter is the tab character (code 9). The default record delimiter is the carriage return character (code 13). You can change these defaults by assigning values to the two delimiter system variables: FldDelimit and RecDelimit. If under Windows, FldDelimit equals 13, a char 10 (line feed) will be appended after the carriage return. Be aware that these variables are used by other commands such as **IMPORT TEXT** for example. Changing them for the Quick Report editor, changes them everywhere in the application.

*qr 4D View area* (3): If 4D View is active for the user, a 4D View external window is created and populated with the results of the current settings of the Quick Report area.

*qr HTML file* (5): An HTML file is created using the template set by **QR SET HTML TEMPLATE**. For detailed information on how the translation is performed, please refer to the Design Reference manual.

If you pass an invalid *area* number, the error -9850 will be generated.
If the value of the destination *type* is incorrect, the error -9852 will be generated.

## Example

The following code sets the destination as being the text file "Mydoc.txt" and executes the Quick Report:

```
QR SET DESTINATION(MyArea;qr text file;"MyDoc.txt")
QR RUN(MyArea)
```

# QR SET DOCUMENT PROPERTY

QR SET DOCUMENT PROPERTY ( area ; property ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| property | Longint | ⇒ | 1 = Printing dialog, 2 = Document unit |
| value | Longint | ⇒ | Value for the property |

## Description

The **QR SET DOCUMENT PROPERTY** command displays the printing dialog or sets the unit used for the document.

In *property*, you can pass the following constants, located in the **QR Document Properties** constant theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr printing dialog | Longint | 1 | Display of the print dialog box:<br>• If value = 0, the print dialog is not displayed prior to printing.<br>• If value = 1, the print dialog is displayed prior to printing (default value). |
| qr unit | Longint | 2 | Document unit:<br>• If value = 0, the document unit is points.<br>• If value = 1, the document unit is centimeters.<br>• If value = 2, the document unit is inches. |

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid value in the *property* or *value* parameter, the corresponding error (-9852 or -9853) is generated.

# QR SET HEADER AND FOOTER

QR SET HEADER AND FOOTER ( area ; selector ; leftTitle ; centerTitle ; rightTitle ; height {; picture {; pictAlignment}} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| selector | Longint | ➡ | 1 = Header, 2 = Footer |
| leftTitle | String | ➡ | Text displayed on the left side |
| centerTitle | String | ➡ | Text displayed in the middle |
| rightTitle | String | ➡ | Text displayed on the right side |
| height | Longint | ➡ | Header or footer height |
| picture | Picture | ➡ | Picture to display |
| pictAlignment | Longint | ➡ | Alignment attribute for the picture |

## Description

The **QR SET HEADER AND FOOTER** command sets the contents and size of the header or footer.

*selector* selects the header or the footer:

- If *selector* is 1, the header is affected;
- If *selector* is 2, the footer is affected.

*leftTitle*, *centerTitle* and *rightTitle* are the values for, respectively, the left, center and right header/footer.

*height* is the height of the header/footer, expressed in the unit selected for the quick report.

*picture* is a picture that will be displayed in the header or footer.

*pictAlignment* is the alignment attribute for the picture passed in *picture*.

- If *pictAlignment* is 1, the picture is aligned to the left.
- If *pictAlignment* is 2, the picture is centered.
- If *pictAlignment* is 3, the picture is aligned to the right.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *selector* value, the error -9852 will be generated.

## Example

The following statement places the title “Center title” in the header for the Quick Report in MyArea and sets the header height to 200 points:

```
QR SET HEADER AND FOOTER(MyArea;1;"";"Center title";"";200)
```

# QR SET HTML TEMPLATE

QR SET HTML TEMPLATE ( area ; template )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| template | Text | ➡ | HTML template |

## Description

The **QR SET HTML TEMPLATE** command sets the HTML template currently used for the Quick Report *area*. The template will be used when building the report in HTML format.

The template uses a set of tags to process the data in order to either retain a layout close to the original report or to adopt your own custom HTML.

**Note:** You first need to call **QR SET DESTINATION** to set the output to HTML file.

## HTML Tags

*<!--#4DQRheader--> ... <!--/#4DQRheader-->*
The HTML contents that are included between these tags come from the column titles. You will typically use these tags to define the title row of the report.

*<!--#4DQRrow--> ... <!--/#4DQRrow-->*
The HTML contents that are included between these tags are repeated for each data row (including detail and subtotal rows).

*<!--#4DQRcol--> ... <!--/#4DQRcol-->*
The HTML contents that are included between these tags are repeated for each data column within a row. The column order will remain identical to the order in the report. When used in conjunction with *<!--#4DQRcol;n--> ... <!--/#4DQRcol;n-->*, the tags *<!--#4DQRcol--> ... <!--/#4DQRcol-->* will only go through the columns whose contents are not inserted using *<!--#4DQRcol;n--> ... <!--/#4DQRcol;n-->*.
For example, in a report that has five columns, you choose to use *<!--#4DQRcol;2--> ... <!--/#4DQRcol;2-->* to insert data from the second column, *<!--#4DQRcol--> ... <!--/#4DQRcol-->* will go, for each row, through columns 1, 3, 4, and 5. These last tags ignore the column whose contents are published using *<!--#4DQRcol;2--> ... <!--/#4DQRcol;2-->*.

*<!--#4DQRcol;n--> ... <!--/#4DQRcol;n-->*
The HTML contents that are included between these tags are extracted from the column in the report whose number is “n”. If, for example, you want to display a different column order in the HTML output for a three-column report, you could use:
*<!--#4DQRrow--> <!--#4DQRcol;3--> ... <!--/#4DQRcol;3--><!--#4DQRcol;2--> ... <!--/#4DQRcol;2--><!--#4DQRcol;1--> ... <!--/#4DQRcol;1--> <!--/#4DQRrow-->*
In this example, the columns are inserted in the opposite order of the report.

*<!--#4DQRfont--> ... <!--/#4DQRfont-->*
The HTML contents that are included between these tags will be assigned the font of the current column or cell. *<!--#4DQRfont-->* will be replaced by an HTML font definition and *<!--/#4DQRfont-->* will be replaced by the matching closing tag (*</font>*).

*<!--#4DQRface--> ... <!--/#4DQRface-->*
The HTML contents that are included between these tags will be assigned the font style of the current column or cell. *<!--#4DQRface-->* will be replaced by an HTML face definition and *<!--#4DQRface-->* will be replaced by the matching closing tag (*</face>*).

*<!--#4DQRbgcolor-->*
This color tag will be replaced by the current color for the current cell.

*<!--#4DQRdata-->*
This tag will be replaced by the current data for the current cell.

*<!--#4DQRlHeader--><!--#4DQRdata--><!--/#4DQRlHeader-->*
*<!--#4DQRcHeader--><!--#4DQRdata--><!--/#4DQRcHeader-->*
*<!--#4DQRrHeader--><!--#4DQRdata--><!--/#4DQRrHeader-->*
These tags will be replaced respectively by the data in the left, center or right header.

*<!--#4DQRlFooter--><!--#4DQRdata--><!--/#4DQRlFooter-->*
*<!--#4DQRcFooter--><!--#4DQRdata--><!--/#4DQRcFooter-->*
*<!--#4DQRrFooter--><!--#4DQRdata--><!--/#4DQRrFooter-->*
These tags will be replaced respectively by the data in the left, center or right footer.

If you pass an invalid *area* number, the error -9850 will be generated.

# QR SET INFO COLUMN

QR SET INFO COLUMN ( area ; colNum ; title ; object ; hide ; size ; repeatedValue ; displayFormat )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| colNum | Longint | ➡ | Column number |
| title | String | ➡ | Title of the column |
| object | Field, Variable | ➡ | Object assigned for that column |
| hide | Longint | ➡ | 0 = displayed, 1 = hidden |
| size | Longint | ➡ | Column size |
| repeatedValue | Longint | ➡ | 0 = not repeated, 1 = repeated |
| displayFormat | String | ➡ | Format for the data |

## Description

### List mode

The **QR SET INFO COLUMN** command sets the parameters of an existing column.

*area* is the reference of the Quick Report area.

*colNum* is the number of the column to modify.

*title* is the title that will be displayed in the header of the column.

*object* is the actual object of the column (variable, field or formula).

*hide* specifies whether the column is shown or hidden:

- If *hide* is 1, the column is hidden;
- If *hide* is 0, the column is shown.

*size* is the size in pixels to assign to the column. If *size* is -1, the size is made automatic.

*repeatedValue* is the status for data repetition. For example, if the value for a field or variable does not change from one record to the other, it may or may not be repeated when they do not change.

- If *repeatedValue* equals 0, values are not repeated.
- If *repeatedValue* equals 1, values are repeated.

*displayFormat* is the display format. Display formats are the 4D formats compatible with the data displayed.

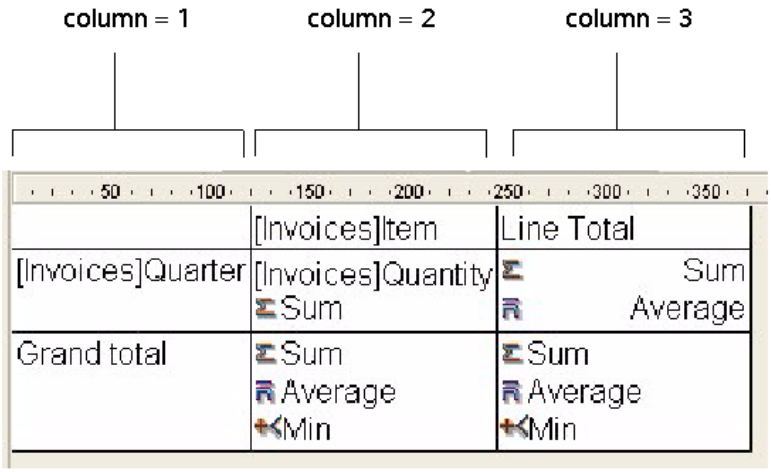
The following statement sets the title of column #1 to Title, sets the contents of the body to Field2, makes the column visible with a width of 150 pixels and sets the format to ###,##.

```
QR SET INFO COLUMN(area;1;"Title";"[Table 1]Field2";0;150;0;"###,##")
```

### Cross-table mode

The **QR SET INFO COLUMN** command allows you to set the same parameters but the reference of the areas to which it applies is different and varies depending on the parameter you want to set.
First of all, the *title*, *hide*, and *repeatedValue* parameters are not used when this command is used in cross-table mode. The value to use for *colNum* varies depending on whether you want to set the column size or the data source and display format.

- Column size
  This is a “visual” attribute, therefore columns are numbered from left to right, as depicted below.

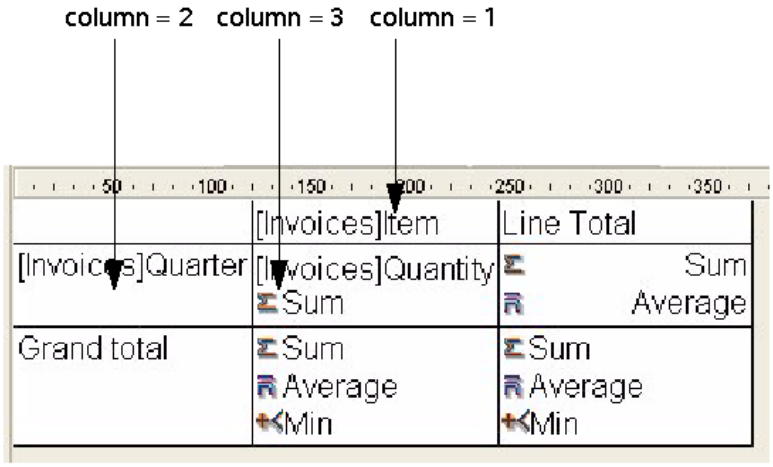



The following statement will set the size to automatic for all the columns in a cross-table report and will leave other elements unchanged:

```
For($i;1;3)
   QR GET INFO COLUMN(qr_area;$i;$title;$obj;$hide;$size;$rep;$format)
   QR SET INFO COLUMN(qr_area;$i;$title;$obj;$hide;0;$rep;$format)
End for
```

You will notice that since you want to alter only the column size, you have to use **QR GET INFO COLUMN** to retrieve the column properties and pass them to **QR SET INFO COLUMN** to leave it unchanged, except for the column size.

- Data source (object) and display format
  In this case the numbering of columns operates as depicted below:

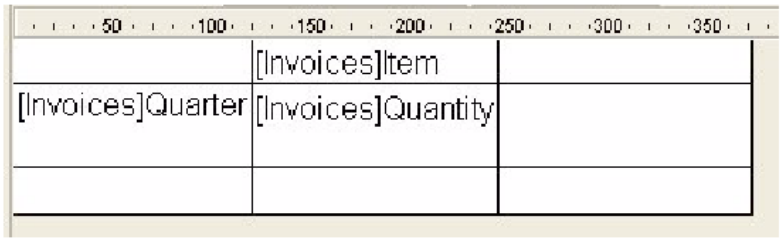



You will notice that not all cells can be addressed using the **QR SET INFO COLUMN** command, the cells that are not numbered above are addressed using **QR SET TOTALS DATA**.

The following code assigns data sources to the three cells required for creating a basic cross-table report:

```
QR SET REPORT TABLE(qr_area;Table(->[Invoices]))
ALL RECORDS([Invoices])
QR SET REPORT KIND(qr_area;2)
QR SET INFO COLUMN(qr_area;1;"";->[Invoices]Item;1;-1;1;"")
QR SET INFO COLUMN(qr_area;2;"";->[Invoices]Quarter;1;-1;1;"")
QR SET INFO COLUMN(qr_area;3;"";->[Invoices]Quantity;1;-1;1;"")
```

This would be the resulting report area:




If you pass an invalid *area* number, the error -9850 will be generated.

If you pass an invalid *colNum* value, the error -9852 will be generated.

# QR SET INFO ROW

QR SET INFO ROW ( area ; row ; hide )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area created |
| row | Longint | ➡ | Row designator |
| hide | Longint | ➡ | 0 = displayed, 1 = hidden |

## Description

The **QR SET INFO ROW** command shows/hides the row whose reference was passed in *row*.

*row* designates which row is affected. You can pass either:

- a positive integer value to designate the corresponding subtotal (break) level,
- one of the following constants from the **QR Rows for Properties** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr detail | Longint | -2 | Detail area of report |
| qr grand total | Longint | -3 | Grand total area |
| qr title | Longint | -1 | Title of report |

*hide* specifies whether the line is shown or hidden:

- If *hide* is 1, the row is hidden;
- If *hide* is 0, the row is shown.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *row* value, the error -9852 will be generated.

## Example

The following statement hides the detail row:

```
QR SET INFO ROW(area;qr detail;1)
```

# QR SET REPORT KIND

QR SET REPORT KIND ( area ; type )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| type | Longint | ⇒ | Type of the report |

## Description

The **QR SET REPORT KIND** command sets the report *type* for the area whose reference was passed in *area*.

- If *type* equals 1, the report type is list.
- If *type* equals 2, the report type is cross-table.

You can also use the constants of the **QR Report Types** theme:

| Constant | Type | Value |
|---|---|---|
| qr cross report | Longint | 2 |
| qr list report | Longint | 1 |

If you set a new type for an existing current report, it removes the previous settings and creates a new empty report, ready to be set.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *type* value, the error -9852 will be generated.

# QR SET REPORT TABLE

QR SET REPORT TABLE ( area ; aTable )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| aTable | Longint | ⇒ | Table number |

## Description

The **QR SET REPORT TABLE** command sets the current table for the report area whose reference was passed in *area* to the table whose number was passed in *aTable*.

It is necessary for a table to be assigned to the report since the report editor will be using the current selection for that table to display the data, perform computations and propagate relations, if needed.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *table* value, the error -9852 will be generated.

# QR SET SELECTION

QR SET SELECTION ( area ; left ; top {; right {; bottom}} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| left | Longint | ➡ | Left boundary |
| top | Longint | ➡ | Top boundary |
| right | Longint | ➡ | Right boundary |
| bottom | Longint | ➡ | Bottom boundary |

## Description

The **QR SET SELECTION** command highlights a cell, a row, a column or the entire *area* as you would with a mouse click. It also lets you deselect the current selection.

*left* is the number of the left boundary. If *left* is 0, the entire row is selected.

*top* is the number of the top boundary. If *top* is 0, the entire column is selected.

*right* is the number of the right boundary.

*bottom* is the number of the bottom boundary.

**Notes:**

- If both *left* and *top* are 0, the entire area is highlighted.
- If you want no selection, pass -1 to *left*, *right*, *top* and *bottom*.

If you pass an invalid *area* number, the error -9850 will be generated.

# QR SET SORTS

QR SET SORTS ( area ; aColumns {; aOrders} )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| aColumns | Real array | ➡ | Columns |
| aOrders | Real array | ➡ | Sort orders |

## Description

The **QR SET SORTS** command sets the sort orders for the columns in the report whose reference is passed in *area*.

*aColumns*: in this array, you need to store the column numbers of columns to which you want to assign a sort order.

*aOrders*: each element of this array must contain the sort orders for the matching column in the *aColumns* array.

- If *aOrders{$i}* is 1, the sort order is ascending.
- If *aOrders{$i}* is -1, the sort order is descending.

### Cross-table mode

In the case of cross-table mode, you cannot have more than two items in the array. You can only sort columns (1) and rows (2). The data (that are the intersection of columns and rows) cannot be sorted.

Here is the code to sort only the rows in the case of a cross-table report:

```
ARRAY REAL($aColumns;1)
$aColumns{1}:=2
ARRAY REAL($aOrders;1)
$aOrders{1}:=-1 `Alphabetic sort for rows
QR SET SORTS(qr_area;$aColumns;$aOrders)
```

If you pass an invalid *area* number, the error -9850 will be generated.

# QR SET TEXT PROPERTY

QR SET TEXT PROPERTY ( area ; colNum ; rowNum ; property ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ➡ | Reference of the area |
| colNum | Longint | ➡ | Column number |
| rowNum | Longint | ➡ | Row number |
| property | Longint | ➡ | Property number |
| value | Longint, String | ➡ | Value for the selected property |

## Description

The **QR SET TEXT PROPERTY** command sets the text attributes for the cell determined by *colNum* and *rowNum*.

*area* is the reference of the Quick Report area.

*colNum* is the number of the cell column.

*rowNum* is the reference of the cell row. You can pass either:

- a positive value designating the corresponding subtotal (break) level,
- one of the constants from the **QR Rows for Properties** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr detail | Longint | -2 | Detail area of report |
| qr footer | Longint | -5 | Page footer |
| qr grand total | Longint | -3 | Grand total area |
| qr header | Longint | -4 | Page header |
| qr title | Longint | -1 | Title of report |

**Note:** When passing -4 or -5 as *rowNum*, you still need to pass a column number in *colNum*, even if it is not used.

**Note:** In cross-table mode, the principle is similar except for the row values, which are always positive.

*property* is the value of the text attribute to assign. You can use the constants of the **QR Text Properties** theme, and the following values can be set:

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_qr font | Longint | 1 | Obsolete since 4D v14R3 (use qr font name) |
| qr alternate background color | Longint | 9 | Alternate background color number |
| qr background color | Longint | 8 | Background color number |
| qr bold | Longint | 3 | Bold style attribute (0 or 1) |
| qr font name | Longint | 10 | Name of font as returned for example by the **FONT LIST** command |
| qr font size | Longint | 2 | Font size expressed in points (9 to 255) |
| qr italic | Longint | 4 | Italic style attribute (0 or 1) |
| qr justification | Longint | 7 | Justification attribute (0 for default, 1 for left, 2 for center or 3 for right) |
| qr text color | Longint | 6 | Color number attribute (Longint) |
| qr underline | Longint | 5 | Underline style attribute (0 or 1) |

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *colNum* number, the error -9852 will be generated.
If you pass an invalid *rowNum* number, the error -9853 will be generated.
If you pass an invalid *property* number, the error -9854 will be generated.

## Example

This method defines several attributes of the first column's title:

```
 //Assigns the Times font:
QR SET TEXT PROPERTY(qr_area;1;-1;qr font name;"Times")
```

```
 //Assigns 10-point font size:
QR SET TEXT PROPERTY(qr_area;1;-1;qr font size;10)
 //Assigns the bold attribute:
QR SET TEXT PROPERTY(qr_area;1;-1;qr bold;1)
 //Assigns the italic attribute:
QR SET TEXT PROPERTY(qr_area;1;-1;qr italic;1)
 //Assigns the underline attribute:
QR SET TEXT PROPERTY(qr_area;1;-1;qr underline;1)
 //Assigns the light green color:
QR SET TEXT PROPERTY(qr_area;1;-1;qr text color;0x0000FF00)
```

# QR SET TOTALS DATA

QR SET TOTALS DATA ( area ; colNum ; breakNum ; operator | value )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| colNum | Longint | ⇒ | Column number |
| breakNum | Longint | ⇒ | Break number |
| operator \| value | Longint, String | ⇒ | Operator value for the cell or Cell content |

## Description

**Note:** This command cannot create a subtotal.

### List Mode

The **QR SET TOTALS DATA** command sets the details of a specific break (total or subtotal).

*area* is the reference of the Quick Report area.

*colNum* is the column number of the cell whose data is going to be set.

*breakNum* is the number of the break whose data will be set (subtotal or grand total). For a Subtotal, *breaknum* is the sort number. For the Grand total, *breaknum* equals -3 or the constant qr grand total of the **QR Operators** theme.

*operator* is an addition of all the operators present in the cell. You can use the constants of the **QR Operators** theme to set the value:

| Constant | Type | Value |
|---|---|---|
| qr average | Longint | 2 |
| qr count | Longint | 16 |
| qr max | Longint | 8 |
| qr min | Longint | 4 |
| qr standard deviation | Longint | 32 |
| qr sum | Longint | 1 |

If *operator* is 0, there is no operator.

*value* is the text to be placed in the cell.

**Note:** Operator/value is mutually exclusive, so you either set an operator or a text.

You can pass the following values:
- # for the value that triggered the break or subtotal
- ##S will be replaced by the sum.
- ##A will be replaced by the Average.
- ##C will be replaced by the Count
- ##X will be replaced by the Max.
- ##N will be replaced by the Min.
- ##D will be replaced by the Standard deviation.
- ##xx, where xx is a column number. This will be replaced by that column's value, using its formatting. If this column does not exist, then it will not be replaced.

### Cross-table Mode

The **QR SET TOTALS DATA** command sets the details of a specific cell.

*area* is the reference of the Quick Report area.

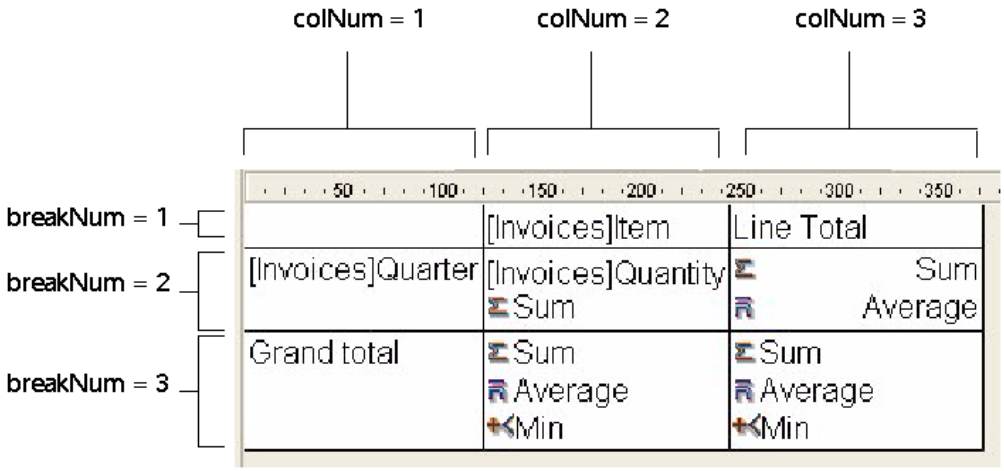
*colNum* is the column number of the cell whose data is going to be set.

*breakNum* is the row number of the cell whose data is going to be set.

*operator* is an addition of all the operators present in the cell. You can use the constants of the **QR Operators** theme to set the value (see above).

*value* is the text to be placed in the cell.

Here is a depiction of how the parameters column and break have to be combined in cross-table mode:




### Supported Types of Data

The types of data that you can pass are of two basic kinds:

- Title
  A title is passed through the parameter *value*. The value is actually a string and can be passed only for the following cells: *colNum=3 breakNum=1* and *colNum=1 breakNum=3*.
- Operator
  An operator or a combination of operators (as described above) can be passed for the following cells:
  *colNum*=2, *breakNum*=2
  *colNum*=3, *breakNum*=2
  *colNum*=2, *breakNum*=3
  Please note that these last two values affect the cell (Column 3; Row 3) as well. If a computation is defined in the cell (Column 2; Row 3), the contents of this cell (Column 2; Row 3) always define the contents of the cell (Column 3; Row 3).

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *colNum* number, the error -9852 will be generated.
If you pass an invalid *breakNum* number, the error -9853 will be generated.

# QR SET TOTALS SPACING

QR SET TOTALS SPACING ( area ; subtotal ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| area | Longint | ⇒ | Reference of the area |
| subtotal | Longint | ⇒ | Subtotal number |
| value | Longint | ⇒ | 0=no space, 32000=inserts a page break, >0=spacing added below the break level, <0=proportional increase |

## Description

The **QR SET TOTALS SPACING** command sets a space below a subtotal row. It applies only to the list mode.

*area* is the reference of the Quick Report area.
*subtotal* is the subtotal level (or break level) that will be affected.

*value* defines the value of the spacing:

- If *value* is 0, no space is added.
- If *value* is 32000, a page break is inserted.
- If *value* is a positive value, it expresses the spacing value in pixels.
- If *value* is a negative value, it expresses the spacing as a percentage of the subtotal row. For example, -100 will set a space of 100% below the subtotal row.

**Note:** If the space below a subtotal row “pushes” the next row onto the following page, there will be no space inserted above the row on that page.

If you pass an invalid *area* number, the error -9850 will be generated.
If you pass an invalid *subtotal*, the error -9852 will be generated.

# Record Locking

- Record Locking
- Get locked records info
- LOAD RECORD
- Locked
- LOCKED BY
- READ ONLY
- Read only state
- READ WRITE
- UNLOAD RECORD

# Record Locking

4D and 4D Server automatically manage databases by preventing multi-user or multi-process conflicts. Two users or two processes cannot modify the same record or object at the same time. However, the second user or process can have read-only access to the record or object at the same time.

There are several reasons for using the multi-user commands:

- Modifying records by using the language.
- Using a custom user interface for multi-user operations.
- Saving related modifications inside a transaction.

There are three important concepts to be aware of when using commands in a multi-processing database:

1. In a process, each table is in either a read-only or a read/write state.
2. Records become locked when they are loaded and unlocked when they are unloaded.
3. A locked record cannot be modified.

As a convention in the following sections, the person performing an operation on the multi-user database is referred to as the **local user**. Other people using the database are referred to as the **other users**. The discussion is from the perspective of the local user. Also, from a multi-process perspective, the process executing an operation on the database is the **current process**. Any other executing process is referred to as **other processes**. The discussion is from the point of view of the current process.

## Locked Records

A locked record cannot be modified by the local user or the current process. A locked record can be loaded, but cannot be modified. A record is locked when one of the other users or processes has successfully loaded the record for modification, or when the record is stacked. Only the user who is modifying the record sees that record as unlocked. All other users and processes see the record as locked, and therefore unavailable for modification. A table must be in a read/write state for a record to be loaded unlocked.

## Read-Only and Read/Write States

Each table in a database is in either a read/write or a read-only state for each user and process of the database. **Read-only** means that records for the table can be loaded but not modified. **Read/write** means that records for the table can be loaded and modified if no other user has locked the record first.

Note that if you change the status of a table, the change takes effect for the next record loaded. If there is a record currently loaded when you change the table's status, that record is not affected by the status change.

### Read-Only State

When a table is read-only and a record is loaded, this record is always locked. In other words, locked records can be displayed, printed, and otherwise used, but they cannot be modified.

Note that the read-only state applies only to editing existing records. A read-only state does not affect the creation of new records. You can still add records to a read-only table using **CREATE RECORD** and **ADD RECORD**, or the menu commands of the Design environment (in this case, the records being created are locked for all other users/processes). Note that the **ARRAY TO SELECTION** command is not affected by the read-only state since it can both create and modify records.

4D automatically sets a table to read-only for commands that do not require write access to records. These commands are: **DISPLAY SELECTION, DISTINCT VALUES, EXPORT DIF, EXPORT SYLK, EXPORT TEXT, _o_GRAPH TABLE, PRINT SELECTION, PRINT LABEL, QR REPORT, SELECTION TO ARRAY, SELECTION RANGE TO ARRAY**.

You can find out the state of a table at any time using the **Read only state** function.

Before executing any of these commands, 4D saves the current state of the table (read-only or read/write) for the current process. After the command has executed, this state is restored.

### Read/Write State

When a table is read/write and a record is loaded, the record will become unlocked if no other user has locked the record first. If the record is locked by another user, the record is loaded as a locked record that cannot be modified by the local user.

A table must be set to read/write and the record loaded for it to become unlocked and thus modifiable.

If a user loads a record from a table in read/write mode, no other users can load that record for modification. However, other users can add records to the table, either through the **CREATE RECORD** and **ADD RECORD** commands or manually in the Design environment.

Read/write is the default state for all tables when a database is opened and a new process is started.

### Changing the Status of a Table

You can use the **READ ONLY** and **READ WRITE** commands to change the state of a table. If you want to change the state of a table in order to make a record read-only or read/write, you must execute the command before this record is loaded. Any record that is already loaded is not affected by the **READ ONLY** and **READ WRITE** commands.

Each process has its own state (read-only or read/write) for each table in the database.

By default, if you do not use the **READ ONLY** command, all tables are in read/write mode.

## Loading, Modifying and Unloading Records

Before the local user can modify a record, the table must be in the read/write state and the record must be loaded and unlocked.

Any of the commands that loads a current record (if there is one) — such as **NEXT RECORD**, **QUERY**, **ORDER BY**, **RELATE ONE**, etc. — sets the record state as locked or unlocked. The record is loaded according to the current state of its table (read-only or read/write) and its availability. A record may also be loaded for a related table by any of the commands that cause an automatic relation to be established.

If a table is in the read-only state for a process or a user, then this table's records are loaded in read-only mode, which means they cannot be modified or deleted by this process or user. This is recommended for viewing or retrieving data because it does not prevent other users or processes from accessing the records of this table in read/write mode if necessary.

If a table is in the read/write state for a process or a user, then any record from this table is also loaded in read/write mode, but only if no other user or process has already locked this record. If a record is successfully loaded in read/write mode, it is unlocked for the current process or user (it can be modified and saved) and is locked for all other users or processes. A table must be put into the read/write state before loading a record for modification and then saving it.

If the record is to be modified, you use the **Locked** function to test whether or not a record is locked by another user. If a record is locked (**Locked** returns True), load the record with the **LOAD RECORD** command and again test whether or not the record is locked. This sequence must be continued until the record becomes unlocked (**Locked** returns False).

When modifications to be made to a record are finished, the record must be released (and therefore unlocked for the other users) with **UNLOAD RECORD**. If a record is not unloaded, it will remain locked for all other users until a different current record is selected. Changing the current record of a table automatically unlocks the previous current record. You need to explicitly call **UNLOAD RECORD** if you do not change the current record. This discussion applies to existing records. When a new record is created, it can be saved regardless of the state of the table to which it belongs.

**Note:** When it is used in a transaction, the **UNLOAD RECORD** command unloads the current record only for the process that manages the transaction. For other processes, the record stays locked as long as the transaction has not been validated (or cancelled).

Use the **LOCKED BY** command to see which user and/or process have locked a record.

**Note:** A good practice is to place all tables in read-only mode when each process is started (using the syntax **READ ONLY(*)**) then put each table in read/write mode only when necessary. Access to tables in read-only mode is faster and more memory-efficient. Moreover, changing the state of a table is optimized in client/server mode because it does not cause any additional network traffic: information is only sent to the server when executing a command that requires adequate access to the table.

## Loops to Load Unlocked Records

The following example shows the simplest loop with which to load an unlocked record:

```
READ WRITE([Customers]) ` Set the table's state to read/write
Repeat ` Loop until the record is unlocked
   LOAD RECORD([Customers]) ` Load record and set locked status
Until(Not(Locked([Customers])))
 ` Do something to the record here
READ ONLY([Customers]) ` Set the table's state to read-only
```

The loop continues until the record is unlocked.

A loop like this is used only if the record is unlikely to be locked by anyone else, since the user would have to wait for the loop to terminate. Thus, it is unlikely that the loop would be used as is unless the record could only be modified by means of a method.

The following example uses the previous loop to load an unlocked record and modify the record:

```
READ WRITE([Inventory])
Repeat ` Loop until the record is unlocked
   LOAD RECORD([Inventory]) ` Load record and set it to locked
Until(Not(Locked([Inventory])))
[Inventory]Part Qty:=[Inventory]Part Qty 1 ` Modify the record
SAVE RECORD([Inventory]) ` Save the record
UNLOAD RECORD([Inventory]) ` Let other users modfiy it
READ ONLY([Inventory])
```

The **MODIFY RECORD** command automatically notifies the user if a record is locked, and prevents the record from being modified. The following example avoids this automatic notification by first testing the record with the **Locked** function. If the record is locked, the user can cancel.

This example efficiently checks to see if the current record is locked for the table [Commands]. If it is locked, the process is delayed by the procedure for one second. This technique can be used both in a multi-user or multi-process situation:

```
Repeat
   READ ONLY([Commands]) ` You do not need read/write right now
   QUERY([Commands])
 ` If the search was completed and some records were returned
   If((OK=1) & (Records in selection([Commands])>0))
      READ WRITE([Commands]) ` Set the table to read/write state
      LOAD RECORD([Commands])
      While(Locked([Commands]) & (OK=1)) `If the record is locked,
 ` loop until the record is unlocked
 ` Who is the record locked by?
         LOCKED BY([Commands];$Process;$User;$SessionUser;$Name)
         If($Process=-1) ` Has the record been deleted?
            ALERT("The record has been deleted in the meantime.")
            OK:=0
         Else
            If($User="") ` Are you in single-user mode
               $User:="you"
            End if
            CONFIRM("The record is already used by "+$User+" in the "+$Name+" Process.")
            If(OK=1) ` If you want to wait for a few seconds
               DELAY PROCESS(Current process;120) ` Wait for a few seconds
               LOAD RECORD([Commands]) ` Try to load the record
            End if
         End if
      End while
      If(OK=1) ` The record is unlocked
         MODIFY RECORD([Commands]) ` You can modify the record
         UNLOAD RECORD([Commands])
      End if
      READ ONLY([Commands]) ` Switch back to read-only
      OK:=1
   End if
Until(OK=0)
```

## Using Commands in Multi-user or Multi-process Environment

A number of commands in the language perform specific actions when they encounter a locked record. They behave normally if they do not encounter a locked record.

Here is a list of these commands and their actions when a locked record is encountered.

- **MODIFY RECORD**: Displays a dialog box stating that the record is in use. The record is not displayed, therefore the user cannot modify the record. In the Design environment, the record is shown in read-only state.

- **MODIFY SELECTION**: Behaves normally except when the user double-clicks a record to modify it. **MODIFY SELECTION** displays dialog box stating that the record is in use and then allows read-only access to the record.
- **APPLY TO SELECTION**: Loads a locked record, but does not modify it. **APPLY TO SELECTION** can be used to read information from the table without special care. If the command encounters a locked record, the record is put into the **LockedSet** system set.
- **DELETE SELECTION**: Does not delete any locked records; it skips them. If the command encounters a locked record, the record is put into the *LockedSet* system set.
- **DELETE RECORD**: This command is ignored if the record is locked. No error is returned. You must test that the record is unlocked before executing this command.
- **SAVE RECORD**: This command is ignored if the record is locked. No error is returned. You must test that the record is unlocked before executing this command.
- **ARRAY TO SELECTION**: Does not save any locked records. If the command encounters a locked record, the record is put into the *LockedSet* system set.
- **GOTO RECORD**: Records in a multi-user/multi-process database may be deleted and added by other users, therefore the record numbers may change. Use caution when directly referencing a record by number in a multi-user database.
- **Sets**: Take special care with sets, as the information that the set was based on may be changed by another user or process.

# Get locked records info

Get locked records info ( aTable ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table where you want to get locked records |
| Function result | Object | ⇐ | Description of locked records (if any) |

## Description

The **Get locked records info** command returns an *object* containing different information about the currently locked record(s) in *aTable*.

The returned object contains a "records" property which is an array of objects:

```
{
   "records": [
      description object,
      (…)
   ]
}
```

Each "description object" array element identifies a locked record in the specified table and has the following properties:

| Property | Type | Description |
|---|---|---|
| contextID | UUID (String) | UUID of the database context responsible for the lock |
| contextAttributes | Object | Object containing information similar to the **LOCKED BY** command but applied to the record, the difference being that **Get locked records info** returns the user name defined in the system and not that of the 4D user, as well as additional information (see below). |
| recordNumber | Longint | Record number of the locked record |

The *contextAttributes* object is made up of the following properties:

| Property | Type | Description |
|---|---|---|
| task_id | Number | Process reference number |
| user_name | String | User name defined by operating system |
| user4d_id | Number | 4D user number(*) |
| host_name | String | Name of host machine |
| task_name | String | Process name |
| client_version | Number | Version of client application |
| *Only when command is executed on 4D Server and if record locking comes from a remote 4D:* | | |
| is_remote_context | Boolean | Indicates whether a remote 4D is the origin of the locking(always *true* since otherwise it is not present) |
| client_uid | UUID (String) | UUID of 4D remote at the origin of the locking |

(*) You can get the 4D user name from the value of *user4d_id* by using the following code:

```
GET USER LIST($arrNames;$arrIDs)
$User4DName:=Find in array($arrIDs;user4d_id)
```

**Note:** The command works only with 4D and 4D Server. It always return an invalid object when called from 4D Remote or a component. However, it can be called in these contexts if the "Execute on server" option is activated. In this case, the object returned will contain, respectively, information about the server or the host database.

## Example

You execute the following code:

```
$vOlocked :=Get locked records info([Table])
```

If two records were locked in the [Table] table, the following object is returned in $vOlocked:


```
{
    "records": [
        {
            "contextID": "A9BB84C0E57349E089FA44E04C0F2F25",
            "contextAttributes": {
                "task_id": 8,
                "user_name": "roland",
                "user4d_id": 1,
                "host_name": "iMac de roland",
                "task_name": "P_RandomLock",
                "client_version": -1342106592
            },
            "recordNumber": 1
        },
        {
            "contextID": "8916338D1B8A4D86B857D92F593CCAC3",
            "contextAttributes": {
                "task_id": 9,
                "user_name": "roland",
                "user4d_id": 1,
                "host_name": "iMac de roland",
                "task_name": "P_RandomLock",
                "client_version": -1342106592
            },
            "recordNumber": 2
        }
    ]
}
```


If the code is executed on a 4D Server and the locking is caused by a remote client machine, the following object is returned in $vOlocked:


```
{
    "records": [
        {
            "contextID": "B0EC087DC2FA704496C0EA15DC011D1C",
            "contextAttributes": {
                "task_id": 2,
                "user_name": "achim",
                "user4d_id": 1,
                "host_name": "achim-pcwin",
                "task_name": "P_RandomLock",
                "is_remote_context": true,
                "client_uid": "0696E66F6CD731468E6XXX581A87554A",
                "client_version": -268364752
            },
            "recordNumber": 1
        }
    ]
}
```

# LOAD RECORD

LOAD RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇨ | Table for which to load record, or Default table, if omitted |

## Description

**LOAD RECORD** loads the current record of *aTable*. If there is no current record, **LOAD RECORD** has no effect.
You can then use the **Locked** function to determine whether you can modify the record:

- If the table is in read-only state, the **Locked** function returns TRUE, and you cannot modify the record.
- If the table is in read/write state but the record was already locked, the record will be read-only, and you cannot modify the record.
- If the table is in read/write state and the record is not locked, you can modify the record in the current process. The **Locked** function returns TRUE for all other users and processes.

**Note:** If the **LOAD RECORD** command is executed after a **READ ONLY**, the record is automatically unloaded and loaded without having to use the **UNLOAD RECORD** command.

Usually, you do not need to use the **LOAD RECORD** command, because commands like **QUERY**, **NEXT RECORD**, **PREVIOUS RECORD**, etc., automatically load the current record.

In multi-user and multi-process environments, when you need to modify an existing record, you must access the table (to which the record belongs) in read/write mode. If a record is locked and not loaded, **LOAD RECORD** allows you to attempt to load the record again at a later time. By using **LOAD RECORD** in a loop, you can wait until the record becomes available in read/write mode.

**Tip:** The **LOAD RECORD** command can be used to reload the current record in the context of an input form. All the data modified are then replaced by their previous values. In this case, the **LOAD RECORD** command carries out a sort of general cancellation of data entry.

# Locked

Locked {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to check for locked current record, or Default table, if omitted |
| Function result | Boolean | ⇐ | Record is locked (TRUE), or Record is unlocked (FALSE) |

## Description

**Locked** tests whether or not the current record of *aTable* is locked. Use this function to find out whether or not the record is locked; then take appropriate action, such as giving the user the choice of waiting for the record to be free or skipping the operation.

If **Locked** returns TRUE, then the record cannot be saved because it is locked by another user or another process, or it is stacked in the current process. In this case, use **LOAD RECORD** to reload the record until **Locked** returns FALSE.

If **Locked** returns FALSE, then the record is unlocked, meaning that the record is locked for all other users. Only the local user or current process can modify and save the record. A table must be in read/write state in order for you to modify the record.

If you try to load a record that has been deleted, **Locked** continues to return TRUE. To avoid waiting for a record that does not exist anymore, use the **LOCKED BY** command. If the record has been deleted, the **LOCKED BY** command returns -1 in the process parameter.

**Note: Locked** returns False when there is no current record in *table*, in other words, when **Record number** returns -1.

During transaction processing, **LOAD RECORD** and **Locked** are often used to test record availability. If a record is locked, it is common to cancel the transaction.

# LOCKED BY

LOCKED BY ( {aTable ;} process ; 4Duser ; sessionUser ; processName )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to check for record locked, or Default table, if omitted |
| process | Longint | ⇐ | Process reference number |
| 4Duser | String | ⇐ | 4D user name |
| sessionUser | String | ⇐ | Name of user that opened work-session |
| processName | String | ⇐ | Process name |

## Description

**LOCKED BY** returns information about the user and process that have locked a record. The process number(*), the user name in the 4D application and in the system as well as the process name are returned in the *process*, *4Duser*, *sessionUser*, and *processName* variables. You can use this information in a custom dialog box to warn the user when a record is locked.

(*) This is the number of the process on the machine where the code that actually locked the record is executed. In the case of a trigger or a method that is executed on the server, the number of the "twin" process on the server machine is returned. In the case of a method that is executed on a remote application, the number of the process on the remote machine is returned.

If the record is not locked, *process* returns 0 and *4Duser*, *sessionUser*, and *processName* return empty strings. If the record you try to load in read/write has been deleted, *process* returns -1 and *4Duser*, *sessionUser*, and *processName* return empty strings.

The *4Duser* parameter returned is the user name from the 4D password system, even if user name is blank. If there is no password system, “Designer” is returned.

The *sessionUser* parameter returned corresponds to the name of the user that opened the session on the client machine (this name is displayed more particularly in the 4D Server administration window for each open process).

# READ ONLY

READ ONLY {( aTable | * )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable \| * | Table, Operator | ⇨ | Table for which to set read-only state, or * for all the tables, or Default table, if omitted |

## Description

**READ ONLY** changes the state of *aTable* to read-only for the process in which it is called. All subsequent records that are loaded are locked, and you cannot make any changes made to them. If the optional * parameter is specified, all tables are changed to read-only state.

Use **READ ONLY** when you do not need to modify the record or records.

**Note:** This command is not retroactive. A record is loaded according to the table's read/write status at the time of loading. To load a record from a read/write table in read-only mode, you must first change the table state to read-only.

# Read only state

Read only state {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to test read-only state, or Default table, if omitted |
| Function result | Boolean | ⮌ | Access to table is read-only (TRUE), or Access to table is read-write (FALSE) |

## Description

This function tests whether or not the state of *aTable* is read-only for the process in which it is called. **Read only state** returns TRUE if the state of *aTable* is read-only. **Read only state** returns FALSE if the state of *aTable* is read/ write.

## Example

The following example tests the state of an [Invoice] table. If the state of the [Invoice] table is read-only, it is set to read/write, and then the current record is reloaded.

```
If(Read only state([Invoice]))
   READ WRITE([Invoice])
   LOAD RECORD([Invoice])
End if
```

**Note:** The invoice record is reloaded to allow the user to modify it. A record that was previously loaded in a read-only state will remain locked until it is reloaded in a read/write state.

# READ WRITE

READ WRITE {( aTable | * )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable \| * | Table, Operator | ⇒ | Table for which to set read-write state, or * for all the tables, or Default table, if omitted |

## Description

**READ WRITE** changes the state of *aTable* to read/write for the process in which it is called. If the optional * parameter is specified, all tables are changed to read/write state.

After a call to **READ WRITE**, when a record is loaded, the record is unlocked if no other user has locked the record. This command does not change the status of the currently loaded record, only that of subsequently loaded records.

The default state for all tables is read/write.

Use **READ WRITE** when you must modify a record and save the changes. Also use **READ WRITE** when you must lock a record for other users, even if you are not making any changes. Setting a table to read/write mode prevents other users from editing that table. However, other users can create new records.

**Note:** This command is not retroactive. A record is loaded according to the table's read/write status at the time of loading. To load a record from a read-only table in read/write mode, you must first change the table state to read/write.

# UNLOAD RECORD

UNLOAD RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to unload record, or Default table, if omitted |

## Description

**UNLOAD RECORD** unloads the current record of *table*.

If the record is unlocked for the local user (locked for the other users), **UNLOAD RECORD** unlocks the record for the other users.

Although **UNLOAD RECORD** unloads it from memory, the record remains the current record. When another record is made the current record, the previous current record is automatically unloaded and therefore unlocked for other users. Always execute this command when you have finished modifying a record and want to make it available to other users, while retaining the record as your current record.

If a record has a large amount of data, picture fields, or external documents (such as 4D Write or 4D Draw documents), you may not want to keep the current record in memory unless you need to modify it. In this case, use the **UNLOAD RECORD** command to keep the current record without having it in memory. You free the memory occupied by the record, but you do not have access to its field values. If you later need access to the values of the record, use the **LOAD RECORD** command.

**Note:** When it is used in a transaction, the **UNLOAD RECORD** command unloads the current record only for the process that manages the transaction. For other processes, the record stays locked as long as the transaction has not been validated (or cancelled).

# Records

- About Record Numbers
- Using the Record Stack
- CREATE RECORD
- DELETE RECORD
- DISPLAY RECORD
- DUPLICATE RECORD
- GOTO RECORD
- Is new record
- Is record loaded
- Modified record
- POP RECORD
- PUSH RECORD
- Record number
- Records in table
- SAVE RECORD
- Sequence number

# About Record Numbers

There are three numbers associated with a record:

- Record number
- Selected record number
- Sequence number

## Record Number

The record number is the absolute/physical record number for a record. A record number is automatically assigned to each new record and remains constant for the record until the record is deleted. Record numbers start at zero. They are not unique because record numbers of deleted records are reused for new records. They also change when the database is compacted or repaired.

## Selected Record Number

The selected record number is the position of the record in the current selection, and so depends on the current selection. If the selection is changed or sorted, the selected record number will probably change. Numbering for the selected record number starts at one (1).

## Sequence Number

The sequence number is a unique non-repeating number that may be assigned to a field of a record (via the **Autoincrement** property, the SQL AUTO_INCREMENT attribute or the **Sequence number** command). It is not automatically stored with each record. It starts by default at 1 and is incremented for each new record that is created. Unlike record numbers, a sequence number is not reused when a record is deleted or when a database is compacted or repaired. Sequence numbers provide a way to have unique ID numbers for records. If a sequence number is incremented during a transaction, the number is not decremented if the transaction is canceled.

**Note:** 4D does not carry out any check when you modify the automatic number internal counter of a table using the **SET DATABASE PARAMETER** command. If you decrement this counter, the new records created may have numbers that have already been assigned.

## Record Number Examples

The following tables illustrate the numbers that are associated with records. Each line in the table represents information about a record. The order of the lines is the order in which records would be displayed in an output form.

- **Data column**: The data from a field in each record. For our example, it contains a person's name.
- **Record Number column**: The record's absolute record number. This is the number returned by the **Record number** function.
- **Selected Record Number column**: The record's position in the current selection. This is the number returned by the **Selected record number** function.
- **Sequence Number column**: The record's unique sequence number. This is the number returned by the **Sequence number** function when the record was created. This number is stored in a field.

### After the Records Are Entered

The first table shows the records after they are entered.

- The default order for the records is by record number.
- The record number starts at 0.
- The selected record number and the sequence number start at 1.

| Data | Record Number | Selected Record Number | Sequence Number |
|---|---|---|---|
| Tess | 0 | 1 | 1 |
| Terri | 1 | 2 | 2 |
| Sabra | 2 | 3 | 3 |
| Sam | 3 | 4 | 4 |
| Lisa | 4 | 5 | 5 |

**Note**: The records remain in the default order after a command changes the current selection without reordering it; for example, after the Show All menu command is chosen in the Design environment, or after the **ALL RECORDS** command is executed.

### After the Records Are Sorted

The next table shows the same records sorted by name.

- The same record number remains associated with each record.
- The selected record numbers reflect the new position of the records in the sorted selection.
- The sequence numbers never change, since they were assigned when each record was created and are stored in the record.

| Data | Record Number | Selected Record Number | Sequence Number |
|---|---|---|---|
| Lisa | 4 | 1 | 5 |
| Sabra | 2 | 2 | 3 |
| Sam | 3 | 3 | 4 |
| Terri | 1 | 4 | 2 |
| Tess | 0 | 5 | 1 |

### After a Record Is Deleted

The following table shows the records after Sam is deleted.

- Only the selected record numbers have changed. Selected record numbers reflect the order in which the records are displayed.

| Data | Record Number | Selected Record Number | Sequence Number |
|---|---|---|---|
| Lisa | 4 | 1 | 5 |
| Sabra | 2 | 2 | 3 |
| Terri | 1 | 3 | 2 |
| Tess | 0 | 4 | 1 |

### After a Record Is Added

The next table shows the records after a new record has been added for Liz.

- A new record is added to the end of the current selection.
- Sam’s record number is reused for the new record.
- The sequence number continues to increment.

| Data | Record Number | Selected Record Number | Sequence Number |
|---|---|---|---|
| Tess | 0 | 1 | 1 |
| Terri | 1 | 2 | 2 |
| Sabra | 2 | 3 | 3 |
| Lisa | 4 | 4 | 5 |
| Liz | 3 | 5 | 6 |

### After the Selection is Changed and Sorted

The following table shows the records after the selection was reduced to three records and then sorted.

- Only the selected record number associated with each record changes.

| Data | Record Number | Selected Record Number | Sequence Number |
|---|---|---|---|
| Sabra | 2 | 1 | 3 |
| Liz | 3 | 2 | 6 |
| Terri | 1 | 3 | 2 |

## Using the Record Stack

---

The **PUSH RECORD** and **POP RECORD** commands allow you to put (“push”) records onto the record stack, and to remove (“pop”) them from the stack.

Each process has its own record stack for each table. 4D maintains the record stacks for you. Each record stack is a last-in-first-out (LIFO) stack. Stack capacity is limited by memory.

**PUSH RECORD** and **POP RECORD** should be used with discretion. Each record that is pushed uses part of free memory. Pushing too many records can cause an out-of-memory or stack full condition.

4D clears the stack of any unpopped records when you return to the menu at the end of execution of your method.

**PUSH RECORD** and **POP RECORD** are useful when you want to examine records in the same file during data entry. To do this, you push the record, search and examine records in the file (copy fields into variables, for example), and finally pop the record to restore the record.

While entering a record, if you have to check a multiple field unique value, use the **SET QUERY DESTINATION** command. This will save you the calls to **PUSH RECORD** and **POP RECORD** that you were making before and after the call to **QUERY** in order to preserve the data entered in the current record. **SET QUERY DESTINATION** allows you to make a query that does not change the selection nor the current record.

# CREATE RECORD

CREATE RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇨ | Table for which to create a new record, or Default table, if omitted |

## Description

**CREATE RECORD** creates a new empty record for *aTable*, but does not display the new record. Use **ADD RECORD** to create a new record and display it for data entry.

**CREATE RECORD** is used instead of **ADD RECORD** when data for the record is assigned with the language. The new record becomes the current record but the current selection is left untouched.

The record exists in memory only until a **SAVE RECORD** command is executed for the table. If the current record is changed (for example, by a query) before the record is saved, the new record is lost.

**Note:** This command does not require *aTable* to be in read/write mode. It can be used even when the table is in read-only mode (see **Record Locking**).

## Example

The following example archives records that are over 30 days old. It does does this by creating new records in an archival table. When the archiving is finished, the records that were archived are deleted from the [Accounts] table:

```
  ` Find records more than 30 days old
QUERY([Accounts];[Accounts]Entered<(Current date-30))
For($vlRecord;1;Records in selection([Accounts])) ` Loop once for each record
   CREATE RECORD([Archive]) ` Create a new archive record
   [Archive]Number:=[Account]Number ` Copy fields to the archive record
   [Archive]Entered:=[Account]Entered
   [Archive]Amount:=[Account]Amount
   SAVE RECORD([Archive]) ` Save the archive record
   NEXT RECORD([Accounts]) ` Move to the next account record
End for
DELETE SELECTION([Accounts]) ` Delete the account records
```

# DELETE RECORD

DELETE RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table where the current record will be deleted, or Default table, if omitted |

## Description

**DELETE RECORD** deletes the current record of *aTable* in the process. If there is no current record for *aTable* in the process, **DELETE RECORD** has no effect. In a form, you can create a Delete Record button instead of using this command.

**Notes:**

- If the current record is unloaded from memory before calling **DELETE RECORD** (for example, subsequent to an **UNLOAD RECORD**), the current selection of *table* is empty after the deletion occurs.
- The **DELETE RECORD** command does nothing when the table is in **READ ONLY** mode, regardless of whether the record to be deleted is locked or not.

Deleting records is a permanent operation and cannot be undone.

If a record is deleted, the record number will be reused when new records are created. Do not use the record number as the record identifier if you will ever delete records from the database.

## Example

The following example deletes an employee record. The code asks the user what employee to delete, searches for the employee's record, and then deletes it:

```
vFind:=Request("Employee ID to delete:") ` Get an employee ID
If(OK=1)
   QUERY([Employee];[Employee]ID =vFind) ` Find the employee
   DELETE RECORD([Employee]) ` Delete the employee
End if
```

# DISPLAY RECORD

DISPLAY RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to display the current record, or Default table, if omitted |

## Description

The **DISPLAY RECORD** command displays the current record of *aTable*, using the current input form. The record is displayed only until an event redraws the window. Such an event might be the execution of an **ADD RECORD** command, returning to an input form, or returning to the menu bar. **DISPLAY RECORD** does nothing if there is no current record.

**DISPLAY RECORD** is often used to display custom progress messages. It can also be used to generate a free-running slide show.

If a form method exists, an On Load event will be generated.

**WARNING:** Do not call **DISPLAY RECORD** from within a Web connection process, because the command will be executed on the 4D Web server machine and not on the Web browser client machine.

## Example

The following example displays a series of records as a slide show:

```
ALL RECORDS([Demo]) ` Select all of the records
FORM SET INPUT([Demo];"Display") ` Set the form to use for display
For($vlRecord;1;Records in selection([Demo])) ` Loop through all of the records
   DISPLAY RECORD([Demo]) ` Display a record
   DELAY PROCESS(Current process;180) ` Pause for 3 seconds
   NEXT RECORD([Demo]) ` Move to the next record
End for
```

# DUPLICATE RECORD

DUPLICATE RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to duplicate the current record, or Default table, if omitted |

## Description

**DUPLICATE RECORD** creates a new record for *aTable* that is a duplicate of the current record. The new record becomes the current record. If there is no current record, then **DUPLICATE RECORD** does nothing. You must use **SAVE RECORD** to save the new record.

**DUPLICATE RECORD** can be executed during data entry. This allows you to create a clone of the currently displayed record. Remember that you must first execute **SAVE RECORD** in order to save any changes made to the original record.

**Compatibility note:** Beginning with version 11 of 4D, this command no longer supports subtables.

# GOTO RECORD

GOTO RECORD ( {aTable ;} record )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table in which to go to the record, or Default table, if omitted |
| record | Longint | ⇒ | Number returned by Record number |

## Description

**GOTO RECORD** selects the specified record of *aTable* as the current record. The *record* parameter is the number returned by the **Record number** function. After executing this command, the record is the only record in the selection.

If *record* is less than the smallest record number in the database or greater than the greatest record number in the database, 4D generates an error message stating that the record number is out of range. If *record* is equal to the record number of a deleted record, 4D returns the error -10503 and the selection becomes empty.

## Example

See the example for **Record number**.

# Is new record

Is new record {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table of the record to examine or Default table if this parameter is omitted |
| Function result | Boolean | ⇐ | True if the record is being created, False otherwise |

## Description

The **Is new record** command returns True when *aTable*'s current record is being created and has not yet been saved in the current process.

**Compatibility Note:** You can obtain the same information by using the existing **Record number** command, and by testing if it returns -3.
However, we strongly advise you to use **Is new record** instead of **Record number** in this case. In fact, the **Is new record** command ensures compatibility with future versions of 4D.

**4D Server:** This command returns a different result for the On Validate form event depending on whether it is executed on 4D in local mode or 4D in remote mode. In local mode, the command returns False (the record is considered as already created). In remote mode, the command returns True because, in this case, the record is already created on the server but the information has not yet been sent to the client.

## Example

The following two statements are identical. The second one is strongly advised so that the code will be compatible with future versions of 4D:

```
If(Record number([Table])=-3) `Not advised
   ...
End if

If(Is new record([Table])) `Strongly advised
   ...
End if
```

# Is record loaded

Is record loaded {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table of the record to examine or Default table if this parameter is omitted |
| Function result | Boolean | ⇐ | True if the record is loaded Otherwise False |

## Description

The **Is record loaded** command returns True if *aTable*'s current record is loaded in the current process.

**4D Server**: In principle, when tables are linked by automatic relations, the current records of related tables are loaded automatically (see **About Relations**). However, for optimization reasons, 4D Server only loads these records when necessary, for example when reading or assigning a field of the related record. As a result, in this context the **Is record loaded** command will return False in remote mode (it returns True in local mode).

## Example

Instead of using the “Next record” or “Previous record” automatic actions, you can write object methods for these buttons to improve their operation. The “Next” button will display the beginning of the selection if the user is at the end of the selection and the “Previous” button will show the end of the selection when the user is at the beginning of the selection.

```
  ` Object method of the  “Previous”  button (without an automatic action)
If(Form event=On Clicked)
   PREVIOUS RECORD([Group])
   If(Not(Is record loaded([Group])))
      GOTO SELECTED RECORD([Group];Records in selection([Group]))
  `Go to the last record in the selection
   End if
End if

  ` Object method of the  “Next”  button (without an automatic action)
If(Form event=On Clicked)
   NEXT RECORD([Group])
   If(Not(Is record loaded([Group])))
      GOTO SELECTED RECORD([Groups];1)
  `Go to the first record in the selection
   End if
End if
```

# Modified record

Modified record {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to test if current record has been modified, or Default table, if omitted |
| Function result | Boolean | ⇐ | Record has been modified (True), or Record has not been modified (False) |

## Description

**Modified record** returns True if the current record of *aTable* has been modified but not saved; otherwise it returns False. This function allows the designer to quickly test whether or not the record needs to be saved. It is especially valuable in input forms to check whether or not to save the current record before proceeding to the next one. This function always returns True for a new record.

Note that this function always returns True in the following contexts:

- the current record is a new record,
- after the execution of the **PUSH RECORD** and **POP RECORD** commands,
- as soon as a value has been assigned to a field of the record, even if it is the same value as the former one. For example, **Modified record** returns True after the following statement is executed:

```
[Table_1]Field_1:=[Table_1]Field_1
```

## Example

The following example shows a typical use for **Modified record**:

```
If(Modified record([Customers]))
   SAVE RECORD([Customers])
End if
```

# POP RECORD

POP RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇨ | Table for which to pop record, or Default table, if omitted |

## Description

**POP RECORD** pops a record belonging to *aTable* from the table's record stack, and makes the record the current record.

If you push a record, change the selection to not include the pushed record, and then pop the record, the current record is not in the current selection. To designate the popped record as the current selection, use **ONE RECORD SELECT**. If you use any commands that move the record pointer before saving the record, you will lose the copy in memory.

## Example

The following example pops the record for the customer off the record stack:

```
POP RECORD([Customers]) ` Pop customer' s record onto stack
```

# PUSH RECORD

PUSH RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table for which to push record, or Default table, if omitted |

## Description

**PUSH RECORD** pushes the current record of *aTable* (and its subrecords, if any) onto the table's record stack. **PUSH RECORD** may be executed before a record is saved.

If you push a record that was unlocked, this record stays locked for all the other processes and users until you pop and unload it.

**Compatibility note:** Beginning with version 11 of 4D, this command no longer supports subtables.

## Example

The following example pushes the record for the customer onto the record stack:

```
PUSH RECORD([Customer]) ` Push customer's record onto stack
```

# Record number

Record number {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return the number of the current record, or Default table, if omitted |
| Function result | Longint | ⇐ | Current record number |

## Description

**Record number** returns the physical record number for the current record of *aTable*. If there is no current record, such as when the record pointer is before or after the current selection, **Record number** returns –1. If the record is a new record that has not been saved, **Record number** returns –3.

Record numbers can change. The record numbers of deleted records are reused.

**4D Server:** This command returns a different result for the On Validate form event depending on whether it is executed on 4D in local mode or 4D in remote mode. In local mode, the command returns a record number (the record is considered as already created). In remote mode, the command returns -3 because, in this case, the record is already created on the server but the information has not yet been sent to the client.

**Note:** It is recommended to use the **Is new record** command to check whether a record is in the process of being created.

## Example

The following example saves the current record number and then searches for any other records that have the same data:

```
$RecNum:=Record number([People]) ` Get the record number
QUERY([People];[People]Last =[People]Last) ` Anyone else with the last name?
 ` Display an alert with the number of people with the same last name
ALERT("There are "+String(Records in selection([People])+" with that name.")
GOTO RECORD([People];$RecNum) ` Go back to the same record
```

# Records in table

Records in table {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return the number of records, or Default table, if omitted |
| Function result | Longint | ⇐ | Total number of records in the table |

## Description

**Records in table** returns the total number of records in aTable. **Records in selection** returns the number of records in the current selection only. If **Records in table** is used within a transaction, records created during the transaction will be taken into account.

## Example

The following example displays an alert that shows the number of records in a table:

```
ALERT("There are "+String(Records in table([People]))+" records in the table.")
```

# SAVE RECORD

SAVE RECORD {{ aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to save the current record, or Default table, if omitted |

## Description

**SAVE RECORD** saves the current record of *aTable* in the current process. If there is no current record, then **SAVE RECORD** is ignored.

You use **SAVE RECORD** to save a record that you created or modified with code. A record that has been modified and validated by the user in a form does not need to be saved with **SAVE RECORD**. A record that has been modified by the user in a form, but has been canceled, can still be saved with **SAVE RECORD**.

If you call the **SAVE RECORD** command when no field has been modified in the record, the command does nothing (the trigger is not called).

Here are some cases where **SAVE RECORD** is required:

- To save a new record created with **CREATE RECORD** or **DUPLICATE RECORD**
- To save data from **RECEIVE RECORD**
- To save a record modified by a method
- To save a record that contains new or modified subrecord data following an **_o_ADD SUBRECORD**, **_o_CREATE SUBRECORD**, or **_o_MODIFY SUBRECORD** command
- During data entry to save the displayed record before using a command that changes the current record
- During data entry to save the current record

You should not execute a **SAVE RECORD** during the On Validate event for a form that has been accepted. If you do, the record will be saved twice.

## Example

The following example is part of a method that reads records from a document. The code segment receives a record, and then, if it is received properly, saves it:

```
RECEIVE RECORD([Customers]) ` Receive record from disk
If(OK=1) ` If the record is received properly…
   SAVE RECORD([Customers]) ` save it
End if
```

# Sequence number

Sequence number {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return the sequence number, or Default table, if omitted |
| Function result | Longint | ⇒ | Sequence number |

## Description

**Sequence number** returns the next sequence number for *aTable*. The sequence number is unique for each table. It is a non-repeating number that is incremented(*) for each new record created for the table.

(*) For optimization reasons, the incrementation is started only at the first call of the **Sequence number** command or of a feature that gets access to the sequence number (see below). In addition, the numbering can be modified using the **SET DATABASE PARAMETER** command. Consequently, the returned value should not be considered as the count of records created in the *aTable*.

By default, the numbering starts at 1. You can change the numbering for a table using the **SET DATABASE PARAMETER** command.

**Note:** If there is no current record and the numbering has been modified via the **SET DATABASE PARAMETER** command, this number is in fact reserved for the next record creation but it will only be returned by the **Sequence number** function when the **SAVE RECORD** command has actually been called.

The **Sequence number** function is useful in the following cases:

- The sequence number needs an increment greater than 1
- The sequence number is part of a code, for example a part number code.

To store the sequence number by means of a method, create a long integer field in the table and assign the sequence number to the field.

The sequence number returned by this function for the *aTable* is the same number as the one generated when you check the **Autoincrement** option for a field of the table using the Structure inspector, or as the one assigned by using the *#N* symbol as the default value for a field of the *table* in a form (see the 4D Design Reference manual).

**Note:** Automatic incrementation can also be set via the SQL AUTO_INCREMENT attribute.

If the sequence number needs to start at a number other than 1, just add the difference to **Sequence number**. For example, if the sequence number must start at 1000, you would use the following statement to assign the number:

```
[Table1]Seq Field :=Sequence number([Table1])+999
```

## Example

The following example is part of a form method. It tests to see if this is a new record; i.e., if the invoice number is an empty string. If it is a new part number, the method assigns an invoice number. The invoice number is formed from two pieces of information: the sequence number, and the operator's ID, which was entered when the database was opened. The sequence number is formatted as a 5-character string:

```
  ` If this is a new part number, create a new invoice number
If([Invoices]Invoice No="")
  ` The invoice number is a string that ends with the operator' s ID.
   [Invoices]Invoice No:=String(Sequence number;"00000")+[Invoices]OpID
End if
```

# Relations


About Relations
CREATE RELATED ONE
GET AUTOMATIC RELATIONS
GET FIELD RELATION
OLD RELATED MANY
OLD RELATED ONE
RELATE MANY
RELATE MANY SELECTION
RELATE ONE
RELATE ONE SELECTION
SAVE RELATED ONE
SET AUTOMATIC RELATIONS
SET FIELD RELATION

# About Relations

The commands in this theme, in particular **RELATE ONE** and **RELATE MANY**, establish and manage the automatic and non-automatic relations between tables. Before using any of the commands in this theme, refer to the 4D Design Reference manual for information about creating relations between tables.

## Using Automatic Table Relations with Commands

Two tables can be related with automatic table relations. In general, when an automatic table relation is established, it loads or selects the related records in a related table. Many operations cause the relation to be established.

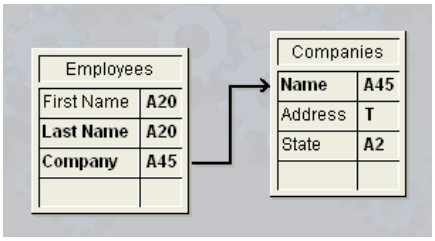
These operations include:

- Data entry
- Listing records on the screen in output forms
- Reporting
- Operations on a selection of records, such as queries, sorts, and applying a formula

To optimize performance, when 4D establishes automatic relations, only one record becomes the current record for a table. For each of the operations listed above, the related record is loaded according to the following principles:

- If a relation selects only one record of a related table, that record is loaded from disk.
- If a relation selects more than one record of a related table, a new selection of records is created for that table, and the first record in that selection is loaded from disk.

For example, using the database structure displayed here, if a record for the [Employees] table is loaded and displayed for data entry, the related record from the [Companies] table is selected and is loaded. Similarly, if a record for the [Companies] table is loaded and displayed for data entry, the related records from the [Employees] table are selected.




In this database structure, the [Employees] table is referred to as the **Many table**, and the [Companies] table is referred to as the **One table**. To remember this concept, think of “there are many employees related to one company” and “each company has many employees.”

Similarly, the Company field in the [Employees] table is referred to as the **Many field**, and the Name field in the [Companies] table is referred to as the **One field**.

It is not always possible to have the related field be unique. For example, the [Companies]Name field may have several company records containing the same value. This non-unique situation can be easily handled by creating a relation, which will always be unique, on another field in the related table. This field could be a company ID field.

The following table lists commands that use automatic relations to load related records during operation of the command. All of the commands will use existing automatic Many-to-One relations. Only those commands with Yes in the One-to-Many established column below will use automatic One-to-Many relations.

| Command | One-to-Many established |
|---|---|
| **ADD RECORD** | Yes |
| **_o_ADD SUBRECORD** | No |
| **APPLY TO SELECTION** | No |
| **DISPLAY SELECTION** | No |
| **EXPORT DIF** | No |
| **EXPORT SYLK** | No |
| **EXPORT TEXT** | No |
| **EXPORT DATA** | No |
| **MODIFY RECORD** | Yes |
| **_o_MODIFY SUBRECORD** | No |
| **MODIFY SELECTION** | Yes (in data entry) |
| **ORDER BY** | No |
| **ORDER BY FORMULA** | No |
| **QUERY BY FORMULA** | Yes |
| **QUERY SELECTION** | Yes |
| **QUERY** | Yes |
| **PRINT LABEL** | No |
| **PRINT SELECTION** | Yes |
| **QR REPORT** | No |
| **SELECTION TO ARRAY** | No |
| **SELECTION RANGE TO ARRAY** | No |

## Using Commands to Establish Table Relations

Automatic relations do not mean that the related record or records for a table will be selected simply because a command loads a record. In some cases, after using a command that loads a record, you must explicitly select the related records by using **RELATE ONE** or **RELATE MANY** if you need to access the related data.

Some of the commands listed in the previous table (such as the query commands) load a current record after the task is completed. In this case, the record that is loaded does not automatically select the records related to it. Again, if you need to access the related data, you must explicitly select the related records by using **RELATE ONE** or **RELATE MANY**.

# CREATE RELATED ONE

CREATE RELATED ONE ( aField )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ⇨ | Many field |

## Description

**CREATE RELATED ONE** performs two actions. If a related record does not exist for *aField* (that is, if a match is not found for the current value of *field*), **CREATE RELATED ONE** creates a new related record.

To save a value in the appropriate field, assign values to the One field from the Many field. Call **SAVE RELATED ONE** to save the new record.

If a related record exists, **CREATE RELATED ONE** acts just like **RELATE ONE** and loads the related record into memory.

# GET AUTOMATIC RELATIONS

GET AUTOMATIC RELATIONS ( one ; many )

| Parameter | Type | | Description |
|---|---|---|---|
| one | Boolean | ⇦ | Status of all Many-to-One relations |
| many | Boolean | ⇦ | Status of all One-to-Many relations |

## Description

The **GET AUTOMATIC RELATIONS** command lets you know if the automatic/manual status of all manual many-to-one and one-to-many relations of the database have been modified in the current process.

- *one*: This parameter returns **True** if a previous calll from the **SET AUTOMATIC RELATIONS** command made all manual many-to-one relations automatic — for example, **SET AUTOMATIC RELATIONS**(True;False).
  This parameter returns **False** if the **SET AUTOMATIC RELATIONS** command has not been called or if its previous execution did not modify manual many-to-one relations — for example, **SET AUTOMATIC RELATIONS**(False;False).
- *many*: This parameter returns **True** if a previous call from the **SET AUTOMATIC RELATIONS** command made all manual one-to-many relations automatic — for example, **SET AUTOMATIC RELATIONS**(True;True).
  This parameter returns **False** if the **SET AUTOMATIC RELATIONS** command has not been called or if its previous execution did not modify manual one-to-many relations — for example, **SET AUTOMATIC RELATIONS**(True;False).

## Example

Refer to the example of the **GET FIELD RELATION** command.

# GET FIELD RELATION

GET FIELD RELATION ( manyField ; one ; many {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| manyField | Field | ⇒ | Starting field of a relation |
| one | Longint | ⇐ | Status of the Many-to-One relation |
| many | Longint | ⇐ | Status of the One-to-Many relation |
| * | Operator | ⇒ | • If passed: one and many return the current status of the relation (values 2 or 3 only) • If omitted (default): one and many can return the value 1 if the relation has not been modified through programming |

## Description

The **GET FIELD RELATION** command lets you find out the automatic/manual status of the relation starting from *manyField* for the current process. You can view any relation, including automatic relations set in the Structure window.

- In *manyField*, pass the name of theMany table field from which the relation whose status you want to find out originates. If no relation originates from the *manyField* field, the *one* et *many* parameters return 0, an error is returned and the system variable OK is set to 0 (see below).
- After the command is executed, the *one* parameter contains a value indicating whether the Many-to-One relation specified is set as automatic:
  0 = There is no relation originating from *manyField*. Syntax error No. 16 (“The field has no relation”) is generated and the system variable OK is set to 0.
  1 = The automatic/manual status of the Many-to-One relation specified is that set by the **Auto Relate One** option in the Relation properties of the Design environment (it has not been modified by programming).
  2 = The Many-to-One relation is manual for the process.
  3 = The Many-to-One relation is automatic for the process.
- After the command is executed, the *many* parameter contains a value indicating whether the One-to-Many relation specified is set as automatic:
  0 = There is no relation originating from *manyField*. Syntax error No. 16 (“The field has no relation”) is generated and the system variable OK is set to 0.
  1 = The automatic/manual status of the One-to-Many relation specified is that set by the **Auto One to Many** option in the Relation properties of the Design environment (it has not been modified by programming).
  2 = The One-to-Many relation is manual for the process.
  3 = The One-to-Many relation is automatic for the process.

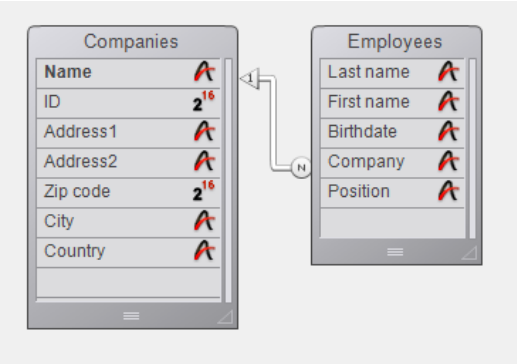
You can compare the values returned in the *one* and *many* parameters with the constants of the “**Relations**” theme:

| Constant | Type | Value |
|---|---|---|
| Automatic | Longint | 3 |
| Manual | Longint | 2 |
| No relation | Longint | 0 |
| Structure configuration | Longint | 1 |

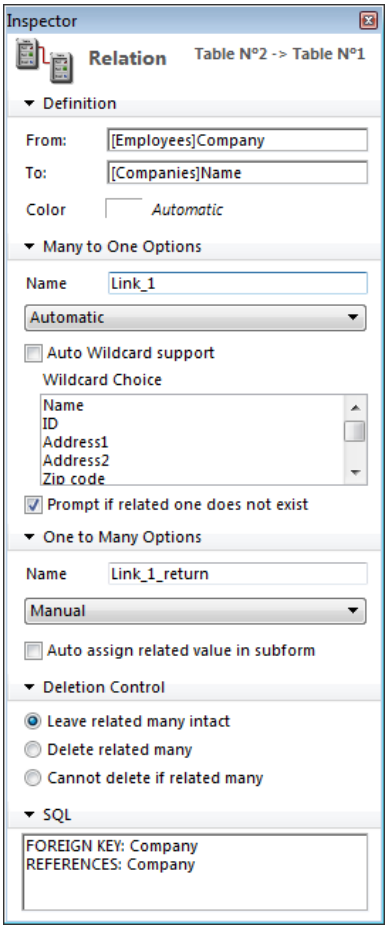
- The optional * parameter lets you “force” the reading of the current status of the relation, even if it has not been modified by programming. In other words, when you pass the * parameter, only the values 2 or 3 can be returned in the *one* and *many* parameters.

## Example

Given the following structure:




The properties of the relation linking the [Employees]Company field to the [Companies]Name field are the following:




The following code illustrates the various possibilities offered by the **GET FIELD RELATION**, **GET AUTOMATIC RELATIONS** and **SET FIELD RELATION**, **SET AUTOMATIC RELATIONS** commands along with their effects:

```
GET AUTOMATIC RELATIONS(one;many) `returns False, False
GET FIELD RELATION([Employees]Company;one;many) `returns 1,1
GET FIELD RELATION([Employees]Company;one;many;*) `returns 3,2

SET FIELD RELATION([Employees]Company;2;0) `changes Many-to-One relation to manual

GET FIELD RELATION([Employees]Company;one;many) `returns 2,1
GET FIELD RELATION([Employees]Company;one;many;*) `returns 2, 2

SET FIELD RELATION([Employees]Company;1;0) `re-establishes the parameters set in Design environment for Many-to-One relation

GET FIELD RELATION([Employees]Company;one;many) `returns 1,1
GET FIELD RELATION([Employees]Company;one;many;*) `returns 3,2

SET AUTOMATIC RELATIONS(True;True) `changes all relations of all tables to automatic

GET AUTOMATIC RELATIONS(one;many) `returns True, True
GET FIELD RELATION([Employees]Company;one;many) `returns 1,1
GET FIELD RELATION([Employees]Company;one;many;*) `returns 3,3
```

# OLD RELATED MANY

OLD RELATED MANY ( aField )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ⇨ | One field |

## Description

**OLD RELATED MANY** operates the same way **RELATE MANY** does, except that **OLD RELATED MANY** uses the old value in the one field to establish the relation.

**Note: OLD RELATED MANY** uses the old value of the many field as returned by the **Old** function. For more information, see the description of the **Old** command.

**OLD RELATED MANY** changes the selection of the related table, and selects the first record of the selection as the current record.

# OLD RELATED ONE

OLD RELATED ONE ( aField )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ⇨ | Many field |

## Description

**OLD RELATED ONE** operates the same way as **RELATE ONE** does, except that **OLD RELATED ONE** uses the old value of *aField* to establish the relation.

**Note: OLD RELATED ONE** uses the old value of the Many field as returned by the **Old** function. For more information, see the description of the **Old** command.

**OLD RELATED ONE** loads the record previously related to the current record. The fields in that record can then be accessed. If you want to modify this old related record and save it, you must call **SAVE RELATED ONE**. Note that there is no old related record for a newly created record.

## System variables and sets

If the command has been executed correctly and if the related records have been loaded, the OK system variable is set to 1. If the user clicked on **Cancel** in the record selection dialog box (that appears when the related record has been modified), the OK variable is set to 0.

# RELATE MANY

RELATE MANY ( oneTable | Field )

| Parameter | Type | | Description |
|---|---|---|---|
| oneTable \| Field | Table, Field | ⇒ | Table to establish all one-to-many relations, or One Field |

## Description

**RELATE MANY** has two forms.

The first form, **RELATE MANY**(oneTable), establishes all One-to-Many relations for *oneTable*. It changes the current selection for each table that has a One-to-Many relation to *oneTable*. The current selections in the Many tables depend on the current value of each related field in the One table. Each time this command is executed, the current selections of the Many tables will be regenerated and the first record of the selection is loaded as the current record..

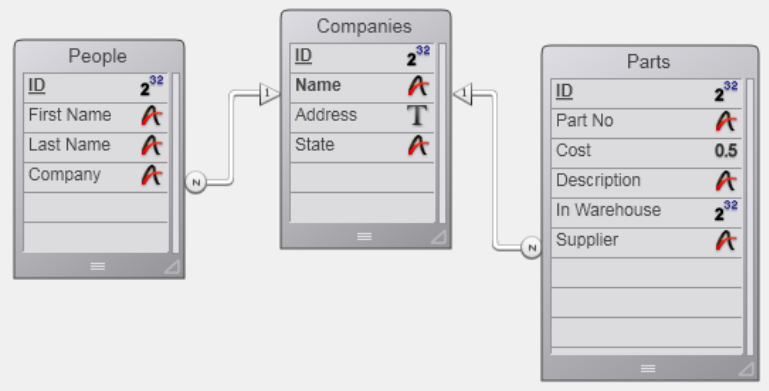
The second form, **RELATE MANY**(oneField), establishes the One-to-Many relation for *oneField*. It changes the current selection and the current record for only those tables that have relations with *oneField*. This means that the related records become the current selection for the Many table.

**Note:** If the current selection in the One table is empty while the **RELATE MANY** command is executed, it has no effect.

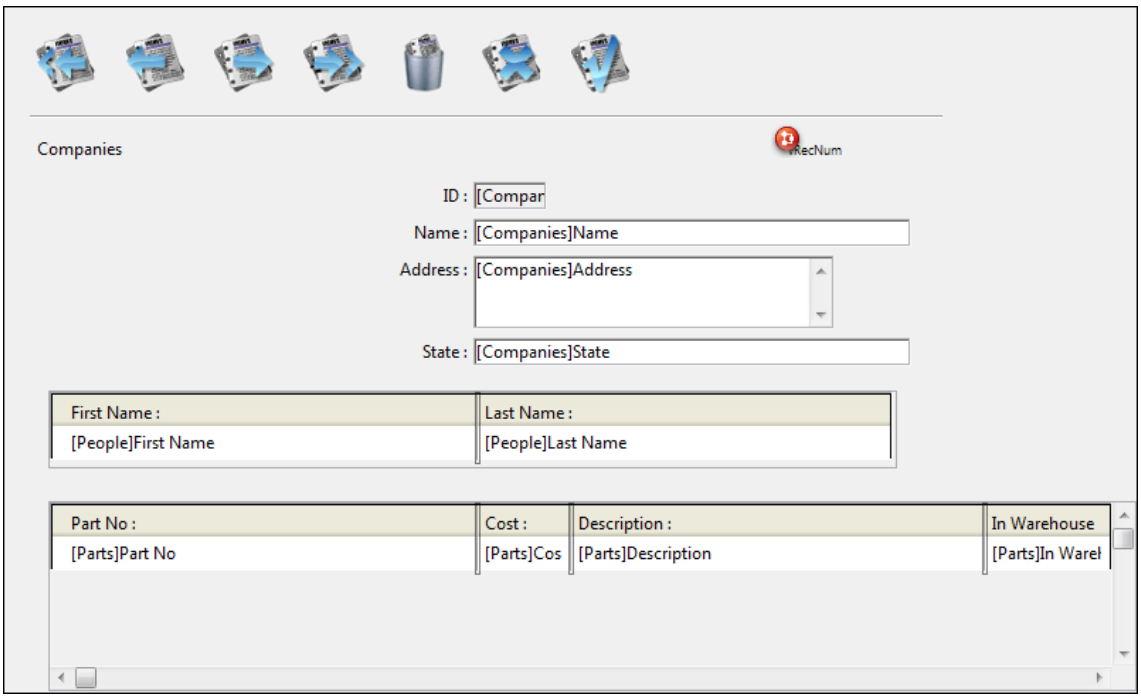
**Note:** This command does not support Object type fields.

## Example

In the following example, three tables are related with automatic relations. Both the [People] table and the [Parts] table have a Many-to-One relation to the [Companies] table.




This form for the [Companies] table will display related records from both the [People] and [Parts] tables.




When the People and Parts forms are displayed, the related records for both the [People] table and the [Parts] table are loaded and become the current selections in those tables.

On the other hand, the related records are not loaded if a record for the [Companies] table is selected programmatically. In this case, you must use the **RELATE MANY** command.

**Notes:**

- When the **RELATE MANY** command is applied to an empty selection, the command is not executed and the selection for the MANY table does not change.
- For the command to work, the foreign key fields (Many fields) must be indexed.

For example, the following method moves through each record of the [Companies] table. An alert box is displayed for each company. The alert box shows the number of people in the company (the number of related [People] records), and the number of parts they supply (the number of related [Parts] records). In the example, the argument to the **ALERT** command is printed on multiple lines for clarity.

Note that the **RELATE MANY** command is needed, even though the relations are automatic.

```
ALL RECORDS([Companies]) ` Select all records in the table
ORDER BY([Companies];[Companies]Name) ` Order records in alphabetical order
For($i;1;Records in table([Companies])) ` Loop once for each record
   RELATE MANY([Companies]Name) ` Select the related records
   ALERT("Company: "+[Companies]Name+Char(13)+"People in company: "
   +String(Records in selection([People]))+Char(13)+"Number of parts they supply: "+String(Records in selection([Parts])))
   NEXT RECORD([Companies]) ` Move to the next record
End for
```

# RELATE MANY SELECTION

RELATE MANY SELECTION ( aField )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ⇒ | Many table field (from which the relation starts) |

## Description

The **RELATE MANY SELECTION** command generates a selection of records in the Many table, based on a selection of records in the One table, and loads the first record of the Many table as the current record.

**Note: RELATE MANY SELECTION** changes the current record for the One table.

## Example

This example selects all invoices made to the customers whose credit is greater than or equal to $1,000. The [Invoices] table field *[Invoices]Customer ID* relates to the [Customer] table field *[Customers]ID Number*.

```
  ` Select the Customers
QUERY([Customers];[Customers]Credit>=1000)
  ` Find all invoices related to any of these customers
RELATE MANY SELECTION([Invoices]Customer ID)
```

# RELATE ONE

RELATE ONE ( manyTable | Field {; choiceField} )

| Parameter | Type | | Description |
|---|---|---|---|
| manyTable \| Field | Table, Field | ➡ | Table for which to establish all automatic relations, or Field with manual relation to one table |
| choiceField | Field | ➡ | Choice field from the one table |

## Description

**RELATE ONE** has two forms.

The first form, **RELATE ONE**(manyTable), establishes all automatic Many-to-One relations for *manyTable* in the current process. This means that for each field in *manyTable* that has an automatic Many-to-One relation, the command will select the related record in each related table. This changes the current record in the related tables for the process.

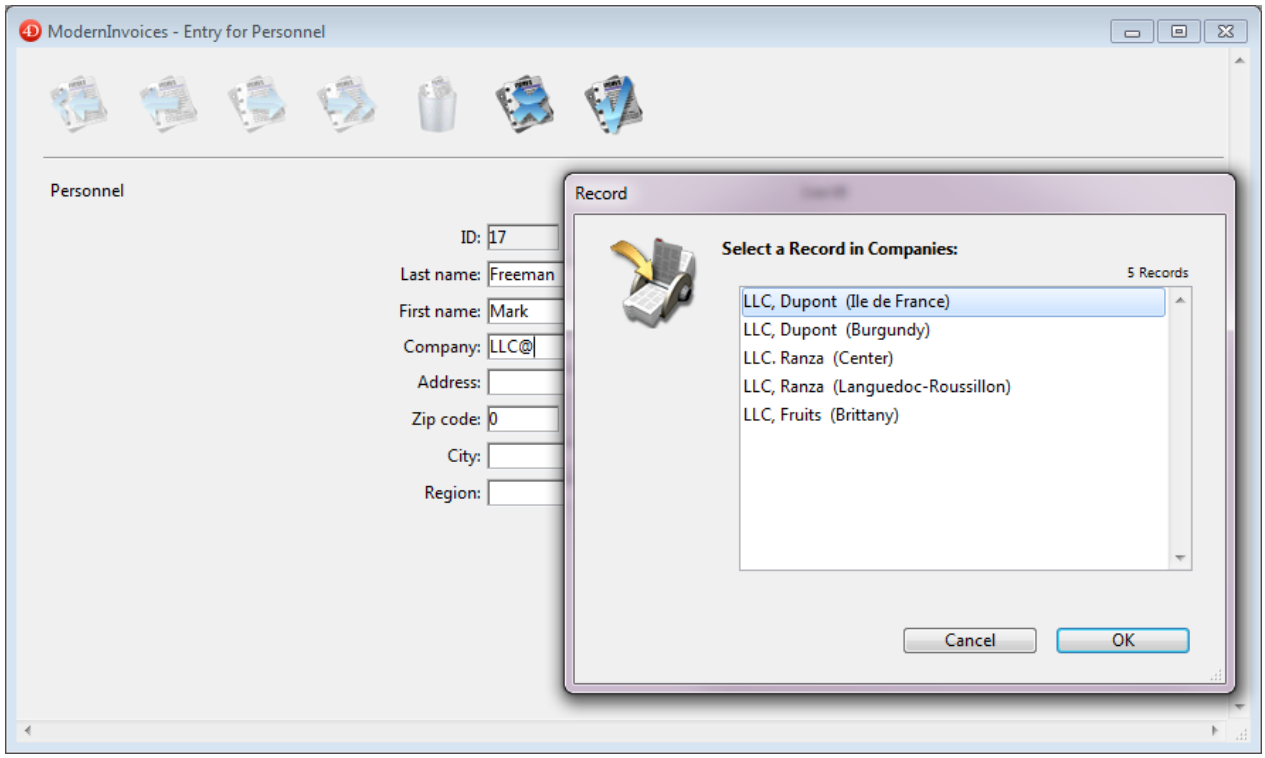
The second form, **RELATE ONE**(manyField{;choiceField}), looks for the record related to *manyField*. The relation does not need to be automatic. If it exists, **RELATE ONE** loads the related record into memory, making it the current record and current selection for its table.

The optional *choiceField* parameter must be a field in the related table. It can only be an Alpha, Text, Numeric, Date, Time, or Boolean field. More specifically, it cannot be a Picture or BLOB type field.

If *choiceField* is specified and more than one record is found in the related table, **RELATE ONE** displays a list of records that match the value in *manyField* so that the user can select a record. In this list, the left column displays related field values, and the right column displays *choiceField* values.

More than one record may be found if *manyField* ends with the wildcard character (@). If there is only one match, the list does not appear.

In the screen below, a record is being entered and a selection list is displayed in the foreground.




The following command is used to make the selection list appear:

```
RELATE ONE([Personnel]Company;[Companies]Region)
```

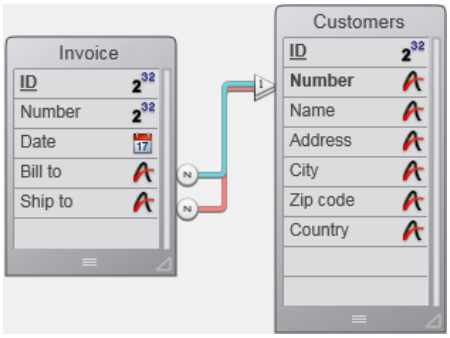
A user entered LLC@ to see a list of companies whose names begin with LLC, as well as their region.

Specifying *choiceField* is the same as specifying a wildcard choice when establishing the table relation. For information about specifying a wildcard choice, refer to the *4D Design Reference* manual.

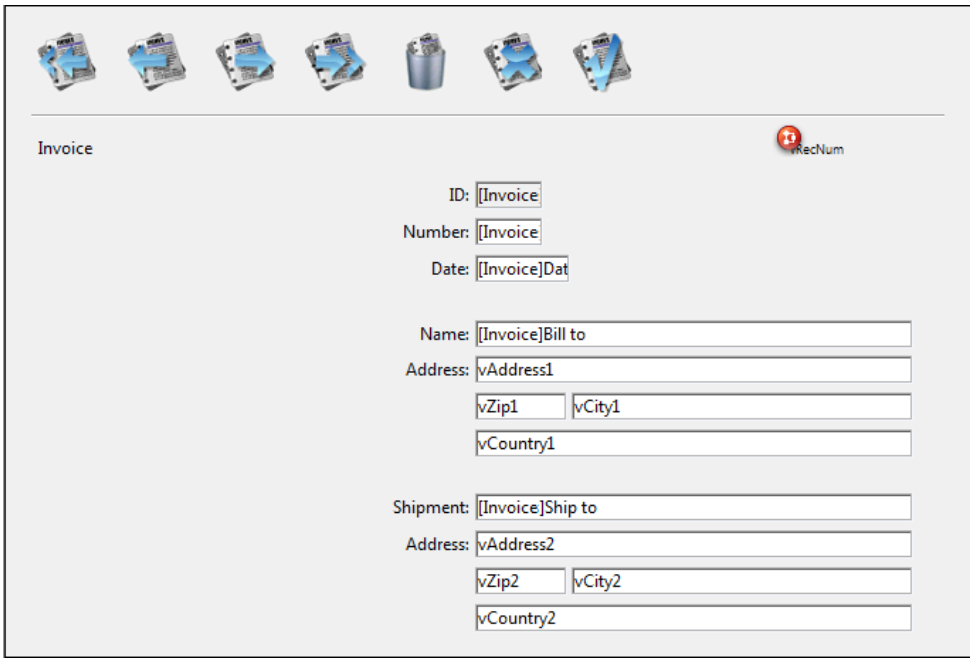
**Note:** This command does not support Object type fields.

## Example

Let's say you have an *[Invoice]* table related to a *[Customers]* table with two non-automatic relations. One relation is from *[Invoice]Bill to* to *[Customers]Number*, and the other relation is from *[Invoice]Ship to* to *[Customers]Number*.




Here is the form for the [Invoice] table displaying the "Bill to" and "Send to" information:




Since both relations are to the same table, *[Customers]*, you cannot obtain the billing and shipment information at the same time. Therefore, displaying both addresses in a form should be performed using variables and calls to **RELATE ONE**. If the *[Customers]* fields were displayed instead, data from only one of the relations would be displayed.

The following two methods are the object methods for the *[Invoice]Bill to* and *[Invoice]Ship to* fields. They are executed when the fields are entered.

Here is the object method for the *[Invoice]Bill to* field:

```
RELATE ONE([Invoice]Bill to)
vAddress1:=[Customers]Address
vCity1:=[Customers]City
vState1:=[Customers]State
vZIP1:=[Customers]ZIP
```

Here is the object method for the *[Invoice]Ship to* field:

```
RELATE ONE([Invoice]Ship to)
vAddress2:=[Customers]Address
vCity2:=[Customers]City
vState2:=[Customers]State
vZIP2:=[Customers]ZIP
```

## System variables and sets

If the command has been executed correctly and if the related records have been loaded, the OK system variable is set to 1. If the user clicked on **Cancel** in the record selection dialog box (that appears when the related record has been modified), the OK variable is set to 0.

# RELATE ONE SELECTION

RELATE ONE SELECTION ( manyTable ; oneTable )

| Parameter | Type | | Description |
|---|---|---|---|
| manyTable | Table | ⇒ | Many table name (from which the relation starts) |
| oneTable | Table | ⇒ | One table name (to which the relation refers) |

## Description

The **RELATE ONE SELECTION** command creates a new selection of records for the table *oneTable*, based on the selection of records in the table *manyTable* and loads the first record of the new selection as the current record.

This command can only be used if there is a relation from *manyTable* to *oneTable*. **RELATE ONE SELECTION** can work across several levels of relations. There can be several related tables between *manyTable* and *oneTable*. The relations can be manual or automatic.

**RELATE ONE SELECTION** uses the "shortest" path to pass from the starting table to the destination table. If there are existing paths of the same size, **RELATE ONE SELECTION** uses the first path found in the creation order of the fields in the starting table.

## Example

The following example finds all the clients whose invoices are due today.

Here is one way of creating a selection in the *[Customers]* table, given a selection of records in the *[Invoices]* table:

```
CREATE EMPTY SET([Customers];"Payment Due")
QUERY([Invoices];[Invoices]DueDate=Current date)
While(Not(End selection([Invoices])))
   RELATE ONE([Invoices]CustID)
   ADD TO SET([Customers];"Payment Due")
   NEXT RECORD([Invoices])
End while
```

The following technique uses **RELATE ONE SELECTION** to accomplish the same result:

```
QUERY([Invoices];[Invoices]DueDate=Current date)
RELATE ONE SELECTION([Invoices];[Customers])
```

**Note:** Since version 11, this code can be written as follows without any loss of performance:

```
QUERY([Customers];[Invoices]DueDate=Current date)
```

# SAVE RELATED ONE

SAVE RELATED ONE ( aField )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ➡ | Many field |

## Description

**SAVE RELATED ONE** saves the record related to *aField*. Execute this command to update a record created with **CREATE RELATED ONE**, or to save modifications to a record loaded with **RELATE ONE**.

**SAVE RELATED ONE** will not save a locked record. When using this command, you must first be sure that the record is unlocked. If the record is locked, the command is ignored, the record is not saved, and no error is returned.

# SET AUTOMATIC RELATIONS

SET AUTOMATIC RELATIONS ( one {; many} )

| Parameter | Type | | Description |
|---|---|---|---|
| one | Boolean | ⇨ | Status of all Many-to-One relations |
| many | Boolean | ⇨ | Status of all One-to-Many relations |

## Description

**SET AUTOMATIC RELATIONS** temporarily changes all the manual relations into automatic relations for the entire database in the current process. The relations stay automatic unless a subsequent call to **SET AUTOMATIC RELATIONS** is made.

- If *one* is true, then all manual Many-to-One relations will become automatic. If *one* is false, all previously changed Many-to-One relations will revert to manual relations.
- The same is true for the *many* parameter, except that manual One-to-Many relations are affected.

This command changes relations set as manual in the Design environment to automatic, just before executing operations that require them to be automatic (such as relational queries and sorts). After the operation is finished, relations can be changed back to manual by calling **SET AUTOMATIC RELATIONS** again.  Any relations set as automatic in the Design environment are not affected by this command.

**Note:** When you pass **True** to the **SET AUTOMATIC RELATIONS** command, the automatic mode is "locked" for all the manual relations during the session. In this case, any calls to the **SET FIELD RELATION** command during the same session are ignored, regardless of whether they are placed before or after **SET AUTOMATIC RELATIONS**. To "unlock" the automatic mode and take the calls to **SET FIELD RELATION** into account, pass **False** to **SET AUTOMATIC RELATIONS**.

## Example

The following example makes all manual Many-to-One relations automatic and reverts any previously changed One-to-Many relations:

```
SET AUTOMATIC RELATIONS(True;False)
```

# SET FIELD RELATION

SET FIELD RELATION ( manyTable | Field ; one ; many )

| Parameter | Type | | Description |
|---|---|---|---|
| manyTable \| Field | Table, Field | ⇒ | Starting table of relations or Starting field of a relation |
| one | Longint | ⇒ | Status of the Many-to-One relation starting from the field or the Many-to-One relations of the table |
| many | Longint | ⇒ | Status of the One-to-Many relation starting from the field or the One-to-Many relations of the table |

## Description

The **SET FIELD RELATION** command sets the automatic/manual status of each relation of the database separately for the current process, regardless of its initial status as specified in the Relation properties window in the Design environment.

In the first parameter, pass a table or field name:

- If you pass a field name (*manyField*), the command only applies to the relation starting from the specified Many field.
- If you pass a table name (*manyTable*), the command applies to all the relations starting from the specified Many table.
- If there is no relation starting from the *manyField* field or *manyTable* table, the syntax error No. 16 (“The field has no relation”) is generated and the system variable OK is set to 0.

In the *one* and *many* parameters, pass the values indicating the changing of the automatic/manual status to be applied respectively to the specified Many-to-One and One-to-Many relation(s). You can use the constants of the “**Relations**” theme:

- Do not modify (0) = Do not modify the current status of the relation(s).
- Structure configuration (1) = Use the configuration set for the relation(s) in the Structure window of the application.
- Manual (2) = Makes the relation(s) manual for the current process.
- Automatic (3) = Makes the relation(s) automatic for the current process.

**Note:** Changes made using this command only apply to the current process. The configuration of the relations set using the options in the Relation properties window is not modified.

**Note:** If you passed **True** to the **SET AUTOMATIC RELATIONS** command during the same session, calls to **SET FIELD RELATION** are ignored, regardless of whether they are placed before or after **SET AUTOMATIC RELATIONS**. To "lock" the automatic mode and take calls to **SET FIELD RELATION** into account, pass **False** to **SET AUTOMATIC RELATIONS**.

## Example

This command makes the management of relations easier in the Quick Report editor. In previous versions of 4D, it was necessary to set all relations as automatic to use them in the editor. Now, the following code allows setting only useful relations as automatic:

```
SET AUTOMATIC RELATIONS(False;False) `Reset of the relations
 `Only the following relations will be used
SET FIELD RELATION([Invoices]Cust_IDt;Automatic;Automatic)
SET FIELD RELATION([Invoice_Row]Invoice_ID;Automatic;Automatic)
QR REPORT([Invoices];Char(1);True;True;True)
```

# Resources

- Resources
- CLOSE RESOURCE FILE
- GET ICON RESOURCE
- Get indexed string
- GET PICTURE RESOURCE
- GET RESOURCE
- Get resource name
- Get resource properties
- Get string resource
- Get text resource
- Open resource file
- RESOURCE LIST
- RESOURCE TYPE LIST
- STRING LIST TO ARRAY
- *_o_ARRAY TO STRING LIST*
- *_o_Create resource file*
- *_o_DELETE RESOURCE*
- *_o_Get component resource ID*
- *_o_SET PICTURE RESOURCE*
- *_o_SET RESOURCE*
- *_o_SET RESOURCE NAME*
- *_o_SET RESOURCE PROPERTIES*
- *_o_SET STRING RESOURCE*
- *_o_SET TEXT RESOURCE*

# Resources

## Compatibility notes about writing resources (4D v13)

As announced with the release of 4D v11 (see section below), mechanisms based on the use of resources files are obsolete. **Commands in the "Resources" theme used to write in resources files must no longer be used; they will be removed from future versions of 4D.** Commands used to read resources are maintained for compatibility.

## Compatibility notes about resource management mechanisms (4D v11)

The management of resources has been modified in 4D beginning with version 11. In conformity with the directions specified by Apple and implemented in the most recent Mac OS versions, the concept of resources in the strictest sense (see definition below) is now obsolete and will be abandoned progressively. New mechanisms have been implemented to support the needs that were previously met by resources: XLIFF files for the translation of character strings, .png picture files, etc. In fact, resource files will be replaced in favor of standard type files. 4D supports this evolution and, beginning with version 11, provides new tools for the management of database translations, while maintaining compatibility with existing systems.

### Compatibility

To maintain compatibility and in order to permit the progressive adaptation of existing applications, the former resource mechanisms will containue to work in 4D v11, with just a few notable differences:

- When present, resource files are still supported by 4D and the principle of the“string of resource files” (successive opening of several resource files) remains valid. The “string of resource files” includes more particularly the .rsr or .4dr files of converted databases (opened automatically) and the custom resource files opened using the commands of this theme.
- However, for reasons related to the evolution of the internal architecture, it is no longer possible to access the resources of the 4D application nor those of the system directly, whether via the commands of this theme or using dynamic references. Certain developers make use of 4D internal resources for their interfaces (for example, resources containing the names of the months or those of the language commands). This practice is now strictly forbidden. In most cases, it is possible to use other means instead of 4D internal resources (constants, language commands, and so on). In order to limit the impact of this modification on existing databases, a substitution system has been implemented, based on the externalization of the resources that are most frequently used. It is nevertheless strongly recommended to change converted databases and to remove any calls to 4D internal resources they may contain.
- Starting with version 11, databases created by 4D will no longer contain .RSR (structure resources) and .4DR (data resources) files by default.

### Current resource management principles

In 4D v11, the notion of “resources” must be understood in the broader sense as “files that are necessary for the translation of application interfaces.” The current architecture of resources is based on a folder, named Resources, that must be located next to the database structure file (.4db or .4dc). In this folder, you need to put all the files that are necessary for the translation or customizing of the application interfaces (picture files, text files, XLIFF files, and so on).
It can also contain any “former generation” resource files of the database (.rsr files). Be careful, these files are not automatically included in the string of resources; they must be opened using standard 4D resource handling commands. 4D uses automatic mechanisms when working with the contents of this folder, in particular for the management of XLIFF files (this point is covered in the *Design Reference* manual). For compatibility reasons, the **Get indexed string** and **STRING LIST TO ARRAY** commands of the "Resources" theme can be used to take advantage of this architecture; however, it is now recommended to use the **Get localized string** command of the "String" theme.

# CLOSE RESOURCE FILE

CLOSE RESOURCE FILE ( resFile )

| Parameter | Type | | Description |
|---|---|---|---|
| resFile | DocRef | ⇨ | Resource file reference number |

## Description

The **CLOSE RESOURCE FILE** command closes the resource file whose reference number is passed in *resFile*.

Even if you have opened the same resource file several times, you need to call **CLOSE RESOURCE FILE** only once in order to close that file.

If you apply **CLOSE RESOURCE FILE** to the 4D application or database resource files, the command detects it and does nothing.

If you pass an invalid resource file reference number, the command does nothing.

Remember to eventually call **CLOSE RESOURCE FILE** for a resource file that you have opened using **Open resource file**. Note that when you quit the application (or open another database), 4D automatically closes all the resource files you opened.

# GET ICON RESOURCE

GET ICON RESOURCE ( resID ; resData {; fileRef} )

| Parameter | Type | | Description |
|---|---|---|---|
| resID | Longint | ➡ | Icon resource ID number |
| resData | Picture | ➡ | Picture field or variable to receive the picture |
| | | ⬅ | Contents of the cicn resource |
| fileRef | DocRef | ➡ | Resource file reference number, or all open resource files, if omitted |

## Compatibility note

*This command is not supported in 64-bit versions of 4D. It returns an error if it is executed in this environment.*

## Description

The **GET ICON RESOURCE** command returns, in the picture field or variable *resData,* the icon stored in the color icon ("cicn") resource whose ID is passed in *resID*.

If the resource is not found, the *resData* parameter is left unchanged and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in *resFile*, the resource is searched for in that file only. If you do not pass *resFile*, the first occurrence of the resource found in the resource files chain is returned.

## Example

The following example loads, in a Picture array, the color icons located in the active 4D application:

```
If(On Windows)
   $vh4DResFile:=Open resource file(Replace string(Application file;".EXE";".RSR"))
Else
   $vh4DResFile:=Open resource file(Application file)
End if
RESOURCE LIST("cicn";$alResID;$asResName;$vh4DResFile)
$vlNbIcons:=Size of array($alResID)
ARRAY PICTURE(ag4DIcon;$vlNbIcons)
For($vlElem;1;$vlNbIcons)
   GET ICON RESOURCE($alResID{$vlElem};ag4DIcon{$vlElem};$vh4DResFile)
End for
```

After this code has been executed, the array looks like this when displayed in a form:

## System variables and sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

# Get indexed string

Get indexed string ( resID ; strID {; resFile} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| resID | Longint | ⇒ | Resource ID number or 'id' attribute of the 'group' element (XLIFF) |
| strID | Longint | ⇒ | String number or 'id' attribute of the 'trans-unit' element (XLIFF) |
| resFile | DocRef | ⇒ | Resource file reference number If omitted: all the XLIFF files or open resource files |
| Function result | String | ⇐ | Value of the indexed string |

## Description

The **Get indexed string** command returns:

- Either one of the strings stored in the string list (“STR#”) resource whose ID is passed in *resID*.
- Or a string stored in an open XLIFF file whose 'id' attribute of the 'group' element is passed in *resID* (see "Compatibility with XLIFF architecture" below).

You pass the number of the string in *strID*. The strings of a string list resource are numbered from 1 to N. To get all the strings (and their numbers) of a string list resource, use the **STRING LIST TO ARRAY** command.

If the resource or the string within the resource is not found, an empty string is returned and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in *resFile*, the resource is searched for in that file only. If you do not pass *resFile*, the first occurrence of the resource found in the resource files chain is returned.

**Note:** A string of a string list resource can contain up to 255 characters.

### Compatibility with XLIFF architecture

The **Get indexed string** command is compatible with the XLIFF architecture of 4D beginning with version 11: the command first looks for values corresponding to *resID* and *strID* in all the open XLIFF files (when the *resFile* parameter is omitted). In this case, *resID* specifies the **id** attribute of the **group** element and *strID* specifies the **id** attribute of the **trans-unit** element. If the value is not found, the command continues searching in the open resources files. For more information about XLIFF architecture in 4D, refer to the Design Reference manual.

## System variables and sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

# GET PICTURE RESOURCE

GET PICTURE RESOURCE ( resID ; resData {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resID | Longint | ⇒ | Resource ID number |
| resData | Field, Variable | ⇒ | Picture field or variable to receive the picture |
| | | ⇐ | Contents of the PICT resource |
| resFile | DocRef | ⇒ | Resource file reference number, or all open resource files, if omitted |

## Description

The **GET PICTURE RESOURCE** command returns in the picture field or variable *resData* the picture stored in the picture ("PICT") resource whose ID is passed in *resID*.

If the resource is not found, the *resData* parameter is left unchanged, and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in *resFile*, the resource is searched for in that file only. If you do not pass *resFile*, the first occurrence of the resource found in the resource files chain is returned.

**Note:** A picture resource can be at least several megabytes in size.

## Example

See example for the **RESOURCE LIST** command.

## System variables and sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

## Error management

If there is not enough memory to load the picture, an error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# GET RESOURCE

GET RESOURCE ( resType ; resID ; resData {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resType | String | ⇒ | 4-character resource type |
| resID | Longint | ⇒ | Resource ID number |
| resData | BLOB | ⇒ | BLOB field or variable to receive the data |
| | | ⇐ | Contents of the resource |
| resFile | DocRef | ⇒ | Resource file reference number, or all open resource files, if omitted |

## Description

The **GET RESOURCE** command returns in the BLOB field or variable *resData* the contents of the resource whose type and ID is passed in *resType* and *resID*.

**Important:** You must pass a 4-character string in *resType*.

If the resource is not found, the *resData* parameter is left unchanged and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in *resFile*, the resource is searched for in that file only. If you do not pass *resFile*, the first occurrence of the resource found in the resource files chain is returned.

**Note:** A resource can be at least several megabytes in size.

## Platform independence

Remember that you are working with Mac OS-based resources. No matter what the platform, internal resource data such as Long Integer is stored using Macintosh byte ordering. On Windows, the data for standard resources (such as string list and pictures resources) is automatically byte swapped when necessary. On the other hand, if you create and use your own internal data structures, it is up to you to byte swap the data you extract from the BLOB (i.e., passing Macintosh byte ordering to a command such as **BLOB to longint**).

## Example

See the example for the **_o_SET RESOURCE** command.

## System variables and sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

## Error management

If there is not enough memory to load the resource, an error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# Get resource name

Get resource name ( resType ; resID {; resFile} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| resType | String | ⇒ | 4-character resource type |
| resID | Longint | ⇒ | Resource ID number |
| resFile | DocRef | ⇒ | Resource file reference number, or all open resource files, if omitted |
| Function result | String | ⇐ | Name of the resource |

## Description

The **Get resource name** command returns the name of the resource whose type is passed in *resType* and whose ID number is passed in *resID*.

If you pass a valid resource file reference number in the parameter *resFile*, the resource is searched for within that file only. If you do not pass the parameter *resFile*, the resource is searched for within the current open resource files.

If the resource does not exist, **Get resource name** returns an empty string.

# Get resource properties

Get resource properties ( resType ; resID {; resFile} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| resType | String | ⇒ | 4-character resource type |
| resID | Longint | ⇒ | Resource ID number |
| resFile | DocRef | ⇒ | Resource file reference number, or all open resource files, if omitted |
| Function result | Longint | ⇐ | Resource attributes |

## Description

The **Get resource properties** command returns the attributes of the resource whose type is passed in *resType* and whose ID number is passed in *resID*.

If you pass a valid resource file reference number in the parameter *resFile*, the resource is searched for within that file only. If you do not pass the parameter *resFile*, the resource is searched for within the current open resource files.

If the resource does not exist, the command returns *0* (zero) and sets the OK variable to *0* (zero).

The numeric value returned by **Get resource properties** must be seen as a bit field value whose bits have special meaning.

## Example

See example for the **Get resource name** command.

## System variables and sets

The OK variable is set to 0 if the resource does not exist; otherwise, it is set to 1.

# Get string resource

Get string resource ( resID {; resFile} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| resID | Longint | ⇒ | Resource ID number |
| resFile | DocRef | ⇒ | Resource file reference number, or all open resource files, if omitted |
| Function result | String | ⇐ | Contents of the STR resource |

## Description

The **Get string resource** command returns the string stored in the string (“STR ”) resource whose ID is passed in *resID*.

If the resource is not found, an empty string is returned and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in *resFile*, the resource is searched for in that file only. If you do not pass *resFile*, the first occurrence of the resource found in the resource files chain is returned.

**Note:** A string resource can contain up to 255 characters.

## Example

The following example displays the contents of the string resource ID=20911, which must be located in at least one of the currently open resource files:

```
ALERT(Get string resource(20911))
```

## System variables and sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

# Get text resource

Get text resource ( resID {; resFile} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| resID | Longint | ⇒ | Resource ID number |
| resFile | DocRef | ⇒ | Resource file reference number, or all open resource files, if omitted |
| Function result | Text | ⇐ | Contents of the TEXT resource |

## Description

The **Get text resource** command returns the text stored in the text (“TEXT”) resource whose ID is passed in *resID*.

If the resource is not found, empty text is returned, and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in *resFile*, the resource is searched for in that file only. If you do not pass *resFile*, the first occurrence of the resource found in the resource files chain is returned.

**Note:** A text resource can contain up to 32,000 characters.

## Example

The following example displays the contents of the text resource ID=20800, which must be located in at least one of the currently open resource files:

```
ALERT(Get text resource(20800))
```

## System variables and sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

# Open resource file

Open resource file ( resFilename {; fileType} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| resFilename | String | ⇒ | Short or long name of resource file, or Empty string for standard Open File dialog box |
| fileType | String | ⇒ | Mac OS file type (4-character string), or Windows file extension (1- to 3-character string), or All files, if omitted |
| Function result | DocRef | ⇐ | Resource file reference number |

## Description

The **Open resource file** command opens the resource file whose name or pathname you pass in *resFileName*.

If you pass a file name, the file should be located in the same folder as the structure file of the database. Pass a pathname to open a resource file located in another folder.

If you pass an empty string in *resFileName*, the Open File dialog box is presented. You can then select the resource file to open. If you cancel the dialog, no resource file is open; **Open resource file** returns a null DocRef and sets the OK variable to 0.

By default, the command opens the resource fork of the file passed as parameter. If it is empty, the command opens the data fork of the file and accesses any resources found there. For more information, refer to the **Current date** section.

If the resource file is opened correctly, **Open resource file** returns its resource file reference number and sets the OK variable to 1. If the resource file does not exist, or if the file you try to open is not a resource file, an error is generated.

- On Macintosh, if you use the Open File dialog box, all files are presented by default. To show a particular type of file, specify the file type in the optional *fileType* parameter.
- On Windows, if you use the Open File dialog box, all files are presented by default. To show a particular type of file, in *fileType*, pass a 1- to 3-character Windows file extension or a Macintosh file type mapped using the **MAP FILE TYPES** command.

Remember to call **CLOSE RESOURCE FILE** for the resource file. Note, however, that when you quit the application (or open another database), 4D automatically closes all the resource files you opened using **Open resource file**.

Unlike the **Open document** command, which opens a document with exclusive read-write access by default, **Open resource file** does not prevent you from opening a resource file already open from within the 4D session. For example, if you try to open the same document twice using **Open document**, an I/O error will be returned at second attempt. On the other hand, if you try to open a resource file already open from within the 4D session, **Open resource file** will return the resource file reference number to the file already open. Even if you open a resource file several times, you need to call **CLOSE RESOURCE FILE** once in order to close that file. Note that this is permitted if the resource file is open from within the 4D session; if you try open a resource file already opened by another application, you will get an I/O error.

**WARNING:**

- It is forbidden to access a 4D application resource file as well as a 4D Desktop merged database resource file.
- Although it is technically possible, you are advised not to use the database structure resource file because your code will not work if the database is compiled and merged with 4D Desktop.
  However, if you access and intend to programmatically add, delete or modify its resources, be sure to test the environment in which you are running. With 4D Server, this will probably lead to serious issues. For example, if you modify a resource on the server machine (via a database method or a stored procedure), you will definitely affect the built-in 4D Server administration service that distributes resources (transparently) to the workstations. Note that with 4D Client, you do not have direct access to the structure file; it is located on the server machine.
- For these reasons, if you use resources, store them in your own files.
- When working with your own resources, do NOT use negative resource IDs; they are reserved for use by the Operating System. Do NOT use resource IDs in the range 0..14,999; this range is reserved for use by 4D. Use the range 15,000..32,767 for your own resources. Remember that once you have opened a resource file, it will be the first file to be searched in the resource files chain. If you store a resource in that file with an ID in the range of system or 4D resources, this resource will be found by commands such as **GET RESOURCE** and also by internal routines of the 4D application. This may be the result you want to achieve, but if you are not sure, do NOT use these ranges, as they may lead to system errors.

- Resource files are highly structured files and cannot accept more than 2,700 resources per file. If you work with files containing a large number of resources, it is a good idea to test that number before adding new resources to a file. See the ***Count resources*** method listed for the **RESOURCE TYPE LIST** command.

After you have opened a resource file, you can analyze the contents of the file using the **RESOURCE TYPE LIST** and **RESOURCE LIST** commands.

## Example 1

The following example tries to open, on Windows, the resource file “MyPrefs.res” located in the database folder:

```
$vhResFile:=Open resource file("MyPrefs";"res ")
```

On Macintosh, the example tries to open the file “MyPrefs”.

## Example 2

The following example tries to open, on Windows. the resource file “MyPrefs.rsr” located in the database folder:

```
$vhResFile:=Open resource file("MyPrefs";"rsr")
```

On Macintosh, the example tries to open the file “MyPrefs”.

## Example 3

The following example displays the Open file dialog box showing all types of files:

```
$vhResFile:=Open resource file("")
```

## Example 4

The following example displays the Open file dialog box showing files created by the **_o_Create resource file** command, using the default file type:

```
$vhResFile:=Open resource file("";"res ")
If(OK=1)
   ALERT("You just opened "+Document+.")
   CLOSE RESOURCE FILE($vhResFile)
End if
```

## System variables and sets

If the resource file is successfully opened, the OK variable is set to 1. If the resource file could not be opened or if the user clicked Cancel in the Open file dialog box, the OK variable is set to 0 (zero).

If the resource file is successfully opened using the Open file dialog box, the Document variable is set to the pathname of the file.

## Error management

If the resource file could not be opened due to a resource or I/O problem, an error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# RESOURCE LIST

RESOURCE LIST ( resType ; resIDs ; resNames {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resType | String | ⇒ | 4-character resource type |
| resIDs | Longint array | ⇐ | Resource ID numbers for resources of this type |
| resNames | String array | ⇐ | Resource names for resources of this type |
| resFile | DocRef | ⇒ | Resource file reference number, or all open resource files, if omitted |

## Description

The **RESOURCE LIST** command populates the arrays *resIDs* and *resNames* with the resource IDs and names of the resources whose type is passed in *resType*.

**Important:** You must pass a 4-character string in *resType*.

If you pass a valid resource file reference number in the optional parameter *resFile*, only the resources from that file are listed. If you do not pass the parameter *resFile*, all resources from the current open resource files are listed.

If you predeclare the arrays before calling **RESOURCE LIST**, you must predeclare *resIDs* as a Longint array and *resNames* as a String or Text array. If you do not predeclare the arrays, the command creates *resIDs* as a Longint array and *resNames* as a Text array.

After the call, you can test the number of resources found by applying the **Size of array** command to the array *resIDs* or *resNames*.

## Example 1

The following example populates the arrays *$alResID* and *$atResName* with the IDs and names of the string list resources present in the structure file of the database:

```
If(On Windows)
   $vhStructureResFile:=Open resource file(Replace string(Structure file;".4DB";".RSR"))
Else
   $vhStructureResFile:=Open resource file(Structure file)
End if
If(OK=1)
   RESOURCE LIST("STR#";$alResID;$atResName;$vhStructureResFile)
End if
```

## Example 2

The following example copies the picture resources present in all currently open resource files into the Picture Library of the database:

```
RESOURCE LIST("PICT";$alResID;$atResName)
Open window(50;50;550;120;5;"Copying PICT resources...")
For($vlElem;1;Size of array($alResID))
   GET PICTURE RESOURCE($alResID{$vlElem};$vgPicture)
   If(OK=1)
      $vsName:=$atResName{$vlElem}
      If($vsName="")
         $vsName:="PICT resID="+String($alResID{$vlElem})
      End if
      ERASE WINDOW
      GOTO XY(2;1)
      MESSAGE("Adding picture  ""+$vsName+""  to the DB Picture library.")
      SET PICTURE TO LIBRARY($vgPicture;$alResID{$vlElem};$vsName)
   End if
End for
CLOSE WINDOW
```

# RESOURCE TYPE LIST

RESOURCE TYPE LIST ( resTypes {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resTypes | String array | ⇐ | List of available resource types |
| resFile | DocRef | ⇒ | Resource file reference number, or all open resource files, if omitted |

## Description

The **RESOURCE TYPE LIST** command populates the array *resTypes* with the resource types of the resources present in the resource files currently open.

If you pass a valid resource file reference number in the optional parameter *resFile*, only the resources from that file are listed. If you do not pass the parameter *resFile*, all the resources from the current open resource files are listed.

You can predeclare the array *resTypes* as a String or Text array before calling **RESOURCE TYPE LIST**. If you do not predeclare the array, the command creates *resTypes* as a Text array.

After the call, you can test the number of resource types found by applying the command **Size of array** to the array *resTypes*.

## Example 1

The following example populates the array *atResType* with the resource types of the resources present in all the resource files currently open:

```
RESOURCE TYPE LIST(atResType)
```

## Example 2

The following example tells you if the Macintosh 4D structure file you are using contains old 4D plug-ins that will need to be updated in order to use the database on Windows:

```
$vhResFile:=Open resource file(Structure file)
RESOURCE TYPE LIST(atResType;$vhResFile)
If(Find in array(atResType;"4DEX")>0)
   ALERT("This database contains old model Mac OS 4D plug-ins."+(Char(13)*2)+
   "You will have to update them for using this database on Windows.")
End if
```

**Note:** The structure file is not the only file where old version plug-ins can be stored. The database can also include a Proc.Ext file.

## Example 3

The following project method returns the number of resources present in a resource file:

```
  ` Count resources project method
  ` Count resources ( Time ) -> Long
  ` Count resources ( DocRef ) -> Number of resources

C_LONGINT($0)
C_TIME($1)

$0:=0
RESOURCE TYPE LIST($atResType;$1)
For($vlElem;1;Size of array($atResType))
   RESOURCE LIST($atResType{$vlElem};$alResID;$atResName;$1)
   $0:=$0+Size of array($alResID)
End for
```

Once this project method is implemented in a database, you can write:

```
$vhResFile:=Open resource file("")
If(OK=1)
   ALERT("The file  ""+Document+""  contains "+String(Count resources($vhResFile))+" resource(s).")
   CLOSE RESOURCE FILE($vhResFile)
End if
```

# STRING LIST TO ARRAY

STRING LIST TO ARRAY ( resID ; strings {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resID | Longint | ⇒ | Resource ID number or 'id' attribute of the 'group' element (XLIFF) |
| strings | String array | ⇐ | Strings from the STR# resource or Strings from the 'group' element (XLIFF) |
| resFile | DocRef | ⇒ | Resource file reference number If omitted: all the XLIFF files or open resources files |

## Description

The **STRING LIST TO ARRAY** command populates the array *strings* with:

- Either the strings stored in the string list ("STR#") resource whose ID is passed in *resID*.
- Or a string stored in an open XLIFF file whose 'id' attribute of the 'group' element is passed in *resID* (see "Compatibility with XLIFF architecture" below).

If the resource is not found, the array *strings* is left unchanged and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in *resFile*, the resource is searched for in that file only. If you do not pass *resFile*, the first occurrence of the resource found in the resource files chain is returned.

Before calling **STRING LIST TO ARRAY**, you can predeclare the array *strings* as a String or Text array. If you do not predeclare the array, the command creates *strings* as a Text array.

**Note:** Each string of a string list resource can contain up to 255 characters.

**Tip:** Limit your use of string list resources to those up to 32K in total size, and a maximum of a few hundred strings per resource.

## Compatibility with XLIFF architecture

The **STRING LIST TO ARRAY** command is compatible with the XLIFF architecture of 4D v11: the command first looks for values corresponding to *resID* and *strID* in all the open XLIFF files (when the *resFile* parameter is omitted) and fills the *strings* array with the corresponding values. In this case, *resID* specifies the **id** attribute of the **group** element and the *strings* array contains all the strings of the element. If the value is not found, the command continues searching in the open resources files.
For more information about XLIFF architecture in 4D, refer to the Design Reference manual.

## System variables and sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

# _o_ARRAY TO STRING LIST

_o_ARRAY TO STRING LIST ( strings ; resID {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| strings | String array | ⇒ | String or Text array (new contents for the STR# resource) |
| resID | Longint | ⇒ | Resource ID number |
| resFile | DocRef | ⇒ | Resource file reference number, or current resource file, if omitted |

## Description

**Compatibility Note:** Commands that write resources are deprecated and should no longer be used.

# _o_Create resource file

_o_Create resource file ( resFilename {; fileType {; *}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| resFilename | String | ⇒ | Short or long name of resource file, or empty string for standard Save File dialog box |
| fileType | String | ⇒ | Mac OS file type (4-character string), or Windows file extension (1- to 3-character string), or Resource ("res " / .RES) document, if omitted |
| * | | ⇒ | If passed = Use data fork |
| Function result | DocRef | ⇐ | Resource file reference number |

## Description

**Compatibility Note:** Commands that write resources are deprecated and should no longer be used.

# _o_DELETE RESOURCE

_o_DELETE RESOURCE ( resType ; resID {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resType | String | ⇒ | 4-character resource type |
| resID | Longint | ⇒ | Resource ID number |
| resFile | DocRef | ⇒ | Resource file reference number, or current resource file, if omitted |

## Compatibility note

Since 4D v13, commands that write resources are deprecated and should no longer be used.

# _o_Get component resource ID

_o_Get component resource ID ( compName ; resType ; originalResNum ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| compName | String | ⇒ | Component name referencing the resource |
| resType | String | ⇒ | Resource type (4 characters), PICT or STR# |
| originalResNum | Longint | ⇒ | Resource original number before component installation |
| Function result | Longint | ⇐ | Current resource number |

## Description

**Compatibility Note:** This command worked with former generation components that are incompatible with version 11 and higher of 4D. It now has no effect and should no longer be used.

# _o_SET PICTURE RESOURCE

_o_SET PICTURE RESOURCE ( resID ; resData {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resID | Longint | ⇒ | Resource ID number |
| resData | Picture | ⇒ | New contents for the PICT resource |
| resFile | DocRef | ⇒ | Resource file reference number, or current resource file, if omitted |

## Description

**Compatibility Note:** Commands that write resources are deprecated and should no longer be used.

# _o_SET RESOURCE

_o_SET RESOURCE ( resType ; resID ; resData {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resType | String | ⇒ | 4-character resource type |
| resID | Longint | ⇒ | Resource ID number |
| resData | BLOB | ⇒ | New contents for the resource |
| resFile | DocRef | ⇒ | Resource file reference number, or current resource file, if omitted |

## Description

**Compatibility Note:** Commands that write resources are deprecated and should no longer be used.

# _o_SET RESOURCE NAME

_o_SET RESOURCE NAME ( resType ; resID ; resName {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resType | String | ⇒ | 4-character resource type |
| resID | Longint | ⇒ | Resource ID number |
| resName | String | ⇒ | New name for the resource |
| resFile | DocRef | ⇒ | Resource file reference number, or current resource file, if omitted |

## Description

**Compatibility Note:** Commands that write resources are deprecated and should no longer be used.

# _o_SET RESOURCE PROPERTIES

_o_SET RESOURCE PROPERTIES ( resType ; resID ; resAttr {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resType | String | ⇒ | 4-character resource type |
| resID | Longint | ⇒ | Resource ID number |
| resAttr | Longint | ⇒ | New attributes for the resource |
| resFile | DocRef | ⇒ | Resource file reference number, or current resource file, if omitted |

## Description

**Compatibility Note:** Commands that write resources are deprecated and should no longer be used.

# _o_SET STRING RESOURCE

_o_SET STRING RESOURCE ( resID ; resData {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resID | Longint | ⇒ | Resource ID number |
| resData | String | ⇒ | New contents for the STR resource |
| resFile | DocRef | ⇒ | Resource file reference number, or current resource file, if omitted |

## Description

**Compatibility Note:** Commands that write resources are deprecated and should no longer be used.

# _o_SET TEXT RESOURCE

_o_SET TEXT RESOURCE ( resID ; resData {; resFile} )

| Parameter | Type | | Description |
|---|---|---|---|
| resID | Longint | ⇒ | Resource ID number |
| resData | String | ⇒ | New contents for the TEXT resource |
| resFile | DocRef | ⇒ | Resource file reference number, or current resource file, if omitted |

## Description

**Compatibility Note:** Commands that write resources are deprecated and should no longer be used.

# Secured Protocol

- GENERATE CERTIFICATE REQUEST
- GENERATE ENCRYPTION KEYPAIR

# GENERATE CERTIFICATE REQUEST

GENERATE CERTIFICATE REQUEST ( privKey ; certifRequest ; codeArray ; nameArray )

| Parameter | Type | | Description |
|---|---|---|---|
| privKey | BLOB | ⇒ | BLOB containing the private key |
| certifRequest | BLOB | ⇐ | BLOB receiving the certificate request |
| codeArray | Longint array | ⇒ | Information code list |
| nameArray | String array | ⇒ | Name list |

## Description

The **GENERATE CERTIFICATE REQUEST** command generates a certificate request at the PKCS format which can be directly used by certificate authorities such as Verisign(R) . The certificate plays an important part in the SSL secured protocol. It is sent to each browser connecting in SSL mode. It contains the “ID card” of the Web site (made from the information entered in the command), as well as its public key allowing the browsers to decrypt the received information. Furthermore, the certificate contains various information added by the certificate authority which guarantees its integrity.

**Note:** For more information on the SSL protocol use with 4D Web server, refer to the **Using TLS Protocol** section.

The certificate request uses keypairs generated with the **GENERATE ENCRYPTION KEYPAIR** command and contains various information. The certificate authority will generate its certificate combining this request with other parameters.

Pass in *privKey* a BLOB containing the private key generated with the **GENERATE ENCRYPTION KEYPAIR** command.

Pass in *certifRequest* an empty BLOB. Once the command has been executed, it contains the certificate request at the PKCS format encoded in base64. You can store the contents directly in a text file suffixed .pem, for example using the **BLOB TO DOCUMENT** command, to submit it to the certificate authority.

**Warning:** The private key is used to generate the request but should NOT be sent to the certificate authority.

The arrays *codeArray* (long integer) and *nameArray* (string) should be filled respectively with the code numbers and the information content required by the certificate authority.

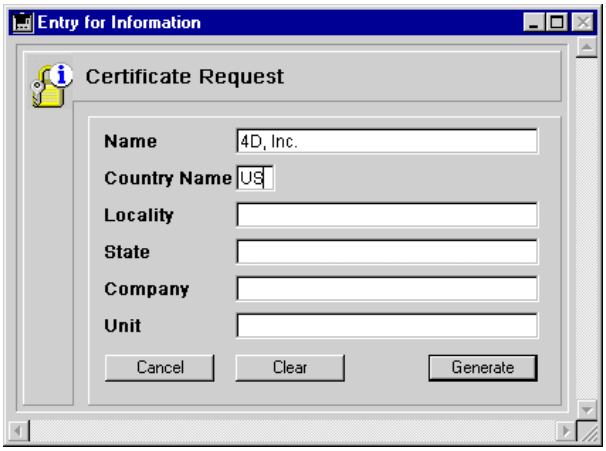
The required codes and names may change according to the certificate authority and the certificate use. However, within a normal use of the certificate (Web server connections via SSL), the arrays should contain the following items:

| Information to provide | codeArray | nameArray (Examples) |
|---|---|---|
| CommonName | 13 | www.4D.com |
| CountryName (two letters) | 14 | US |
| LocalityName | 15 | San Jose |
| StateOrProvinceName | 16 | California |
| OrganizationName | 17 | 4D, Inc. |
| OrganizationUnit | 18 | Web Administrator |

The code and information content entering order does not matter, however the two arrays must be synchronized: if the third item of the *codeArray* contains the value *15* (locality name), the *nameArray* third item should contain this information, in our example San Jose.

## Example

A “Certificate request” form contains the six fields necessary for a standard certificate request. The **Generate** button creates a document on disk containing the certificate request. The “Privatekey.txt” document containing the private key (generated with the **GENERATE ENCRYPTION KEYPAIR** command) should be on the disk:




Here is the **Generate** button method:

```
   ` bGenerate Object Method

C_BLOB($vbprivateKey;$vbcertifRequest)
C_LONGINT($tableNum)
ARRAY LONGINT($tLCodes;6)
ARRAY STRING(80;$tSInfos;6)

$tableNum:=Table(Current form table)
For($i;1;6)
   $tSInfos{$i}:=Field($tableNum;$i)->
   $tLCodes{$i}:=$i+12
End for
If(Find in array($tSInfos;"")#-1)
   ALERT("All fields should be filled.")
Else
   ALERT("Select your private key.")
   $vhDocRef:=Open document("")
   If(OK=1)
      CLOSE DOCUMENT($vhDocRef)
      DOCUMENT TO BLOB(Document;$vbprivateKey)
      GENERATE CERTIFICATE REQUEST($vbPrivateKey;$vbcertifRequest;$tLCodes;$tSInfos)
      BLOB TO DOCUMENT("Request.txt";$vbcertifRequest)
   Else
      ALERT("Invalid private key.")
   End if
End if
```

# GENERATE ENCRYPTION KEYPAIR

GENERATE ENCRYPTION KEYPAIR ( privKey ; pubKey {; length} )

| Parameter | Type | | Description |
|---|---|---|---|
| privKey | BLOB | ⇐ | BLOB to contain the private key |
| pubKey | BLOB | ⇐ | BLOB to contain the public key |
| length | Longint | ⇒ | Key length (bits) [386...2048] Default value = 512 |

## Description

The **GENERATE ENCRYPTION KEYPAIR** command generates a new pair of RSA keys. The security system offered in 4D is based on keys designed to encrypt/decrypt information. They can be used within the TLS/SSL protocol, with 4D Web server (encryption and secured communications) and in all databases (for data encryption).

Once the command has been executed, the BLOBs passed in *privKey* and *pubKey* parameters contain a new pair of encryption keys.

The optional parameter *length* can be used to set the key size (in bits). The larger the key, the more difficult it is to break the encryption code.

However, large keys require longer execution or reply time, especially within a secured connection.

By default (if the *length* parameter is omitted), the generated key size is set to 512 bits, which is a good compromise for the security/efficiency ratio. To increase the security factor, you can change keys more often, for example every six months.You can generate 2048 bits keys to increase the encryption security but the Web application connections will be slowed down.

This command will generate keys in PKCS format encoded in base64, which means that their content can be copied/pasted in an email without any change. Once the pair of keys has been generated, a text document in PEM format can be produced (using the **BLOB TO DOCUMENT** command for example) and the keys can be stored in a safe place.

**Warning:** The private key should always be kept secret.

## About RSA, private key and public key

The RSA cipher used by **GENERATE ENCRYPTION KEYPAIR** is based on a double key encryption system: a private key and a public key. As indicated by its name, the public key can be given to a third person and used to decrypt information. The public key is matched with a unique private key, used to encrypt the information. Thus, the private key is used for encryption; the public key for decryption (or vice versa). The information encrypted with one key can only be decrypted with the other one.

The TLS/SSL protocol encryption functionalities are based on this principle, the public key being included in the certificate sent to the browsers (for more information, see the section **Using TLS Protocol**).

This encryption mode is also used by the first syntax of the **ENCRYPT BLOB** and **DECRYPT BLOB** commands. The public key should be confidentially published.

It is possible to mix the public and private keys from two persons to encrypt information so that the recipient is the only person to be able to decrypt them and the sender is the only person to have encrypted them. This principle is given by the second syntax of the two **ENCRYPT BLOB** and **DECRYPT BLOB** commands.

## Example

See example for the **ENCRYPT BLOB** command.

# Selection

- ALL RECORDS
- APPLY TO SELECTION
- Before selection
- CREATE SELECTION FROM ARRAY
- DELETE SELECTION
- DISPLAY SELECTION
- Displayed line number
- End selection
- FIRST RECORD
- GET HIGHLIGHTED RECORDS
- GOTO SELECTED RECORD
- HIGHLIGHT RECORDS
- LAST RECORD
- MOBILE Return selection
- MODIFY SELECTION
- NEXT RECORD
- ONE RECORD SELECT
- PREVIOUS RECORD
- Records in selection
- REDUCE SELECTION
- SCAN INDEX
- Selected record number
- TRUNCATE TABLE

# ALL RECORDS

ALL RECORDS {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to select all records, or Default table, if omitted |

## Description

**ALL RECORDS** selects all the records of *aTable* for the current process. **ALL RECORDS** makes the first record the current record and loads the record from disk. **ALL RECORDS** returns the records to the default record order, which is the order in which the records are stored on disk.

## Example

The following example displays all the records from the [People] table:

```
ALL RECORDS([People]) ` Select all the records in the table
DISPLAY SELECTION([People]) ` Display records in output form
```

# APPLY TO SELECTION

APPLY TO SELECTION ( aTable ; statement )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table for which to apply statement |
| statement | Statement | ➡ | One line of code or a method |

## Description

**APPLY TO SELECTION** applies *statement* to each record in the current selection of *aTable*. The *statement* can be a statement or a method. If *statement* modifies a record of *aTable*, the modified record is saved. If *statement* does not modify a record, the record is not saved. If the current selection is empty, **APPLY TO SELECTION** has no effect. If the relation is automatic, the *statement* can contain a field from a related table.

**APPLY TO SELECTION** can be used to gather information from the selection of records (for example, a total), or to modify a selection (for example, changing the first letter of a field to uppercase). If this command is used within a transaction, all changes can be undone if the transaction is canceled.

**4D Server:** The server does not execute any of the commands that may be passed in *statement*. Every record in the selection will be sent back to the local workstation to be modified.

The progress thermometer is displayed while **APPLY TO SELECTION** is executing. To hide it, use **MESSAGES OFF** prior to the call to **APPLY TO SELECTION**. If the progress thermometer is displayed, the user can cancel the operation.

## Example 1

The following example changes all the names in the table [Employees] to uppercase:

```
APPLY TO SELECTION([Employees];[Employees]Last Name:=Uppercase([Employees]Last Name))
```

## Example 2

If a record is locked during execution of **APPLY TO SELECTION** and that record is modified, the record will not be saved. Any locked records that are encountered are put in a set called *LockedSet*. After **APPLY TO SELECTION** has executed, test *LockedSet* to see if any records were locked. The following loop will execute until all records have been modified:

```
Repeat
   APPLY TO SELECTION([Employees];[Employees]Last Name:=Uppercase([Employees]Last Name))
   USE SET("LockedSet") ` Select only locked records
Until(Records in set("LockedSet")=0) ` Done when there are no locked records
```

## Example 3

This example uses a method:

```
ALL RECORDS([Employees])
APPLY TO SELECTION([Employees];M_Cap)
```

## System variables and sets

If the user clicks the Stop button in the progress thermometer, the OK system variable is set to 0. Otherwise, the OK system variable is set to 1.

# Before selection

Before selection {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to test if record pointer is before the first selected record, or Default table, if omitted |
| Function result | Boolean | ⇐ | Yes (TRUE) or No (FALSE) |

## Description

**Before selection** returns TRUE when the current record pointer is before the first record of the current selection of *table*.**Before selection** is commonly used to check whether or not **PREVIOUS RECORD** has moved the current record pointer before the first record. If the current selection is empty, **Before selection** returns TRUE.

To move the current record pointer back into the selection, use **FIRST RECORD**, **LAST RECORD** or **GOTO SELECTED RECORD**. **NEXT RECORD** does not move the pointer back into the selection.

**Before selection** also returns TRUE in the first header when a report is being printed with **PRINT SELECTION** or from the Print menu. You can use the following code to test for the first header and print a special header for the first page:

```
  ` Method of a form being used as output form for a summary report
$vpFormTable:=Current form table
Case of
  ` ...
  :(Form event=On Header)
  ` A header area is about to be printed
     Case of
        :(Before selection($vpFormTable->))
  ` Code for the first break header goes here
  ` ...
     End case
End case
```

## Example

This form method is used during the printing of a report. It sets a variable, *vTitle*, to print in the Header area on the first page:

```
  ` [Finances];"Summary" Form Method
Case of
  ` ...
  :(Form event=On Header)
     Case of
        :(Before selection([Finances))
           vTitle:="Corporate Report 1997" ` Set the title for the first page
        Else
           vTitle:="" ` Clear the title for all other pages
     End case
End case
```

# CREATE SELECTION FROM ARRAY

CREATE SELECTION FROM ARRAY ( aTable ; recordArray {; selectionName} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to create the selection |
| recordArray | Longint, Boolean array | ⇒ | Array of record numbers, or Array of booleans (True = the record is in the selection, False = the record is not in the selection) |
| selectionName | String | ⇒ | Name of the named selection to create, or Apply the command to the current selection if the parameter is omitted |

## Description

The **CREATE SELECTION FROM ARRAY** command creates the named selection *selectionName* from:

- either an array of absolute record numbers *recordArray* from *aTable*,
- or an array of Booleans. In this case, the values of the array indicate the belonging (**True**) or not (**False**) of each record in *table* to *selectionName*.

If you don’t pass *selectionName* or if you pass an empty string, the command will be applied to the current selection, which will then be updated.

When you use a Longint array with this command, all the numbers of the array represent the list of record numbers in *selectionName*. If a number is incorrect (record not created), error -10503 is generated.

**Note:** Be careful, you must make sure that the array does not contain any lines that have the same value, otherwise the resulting selection will be incorrect.

When you use a Boolean array with this command, the Xth element of the array indicates if the record number X is (**True**) or is not (**False**) in *selectionName*. The number of elements in *recordArray* must be equal to the number of records in *table*. If the array is smaller than the number of records, only the records defined by the array can make up the selection.

**Note:** With an array of booleans, the command uses elements from numbers 0 to X-1.

**Warning:** A named selection is created and loaded into memory. Therefore, make sure that you have enough memory before executing this command.

## Error management

If a record number is invalid (record not created), the error -10503 is generated. You can intercept this error using a method installed by the **ON ERR CALL** command.

# DELETE SELECTION

DELETE SELECTION {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to delete the current selection, or Default table, if omitted |

## Description

**DELETE SELECTION** deletes the current selection of records from *aTable*. If the current selection is empty, **DELETE SELECTION** has no effect. After the records are deleted, the current selection is empty. Records that are deleted during a transaction are locked to other users and other processes until the transaction is validated or canceled.

**Warning:** Deleting a selection of records is a permanent operation, and cannot be undone.

Unchecking the **Records definitively deleted** option in the table Inspector allows you to increase the speed of deletions when **DELETE SELECTION** is used (see **Records definitively deleted** in the *Design Reference* manual).

## Example 1

The following example displays all the records from the [People] table and allows the user to select which ones to delete. The example has two sections. The first is a method to display the records. The second is an object method for a Delete button. Here is the first method:

```
ALL RECORDS([People]) ` Select all records
FORM SET OUTPUT([People];"Listing") ` Set the form to list the records
DISPLAY SELECTION([People]) ` Display all records
```

The following is the object method for the Delete button, which appears in the Footer area of the output form. The object method uses the records the user selected (the *UserSet*) to delete the selection. Note that if the user did not select any records, **DELETE SELECTION** has no effect.

```
 ` Confirm that the user really wants to delete the records
CONFIRM("You selected "+String(Records in set("UserSet"))+" people to delete."+Char(13)+"Click OK to delete them.")
If(OK=1)
   USE SET("UserSet") ` Use the records chosen by the user
   DELETE SELECTION([People]) ` Delete the selection of records
End if
ALL RECORDS([People]) ` Select all records
```

## Example 2

If a locked record is encountered during the execution of **DELETE SELECTION**, that record is not deleted. Any locked records are put into a set called *LockedSet*. After **DELETE SELECTION** has executed, you can test the *LockedSet* to see if any records were locked. The following loop will execute until all the records have been deleted:

```
Repeat ` Repeat for any locked records
   DELETE SELECTION([ThisTable])
   If(Records in set("LockedSet")#0) ` If there are locked records
      USE SET("LockedSet") ` Select only the locked records
   End if
Until(Records in set("LockedSet")=0) ` Until there are no more locked records
```

# DISPLAY SELECTION

DISPLAY SELECTION ( {aTable}{; selectMode}{; enterList}{; *}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to display, or Default table, if omitted |
| selectMode | Longint | ⇒ | Selection mode |
| enterList | Boolean | ⇒ | Authorize Enter in list option |
| * | | ⇒ | Use output form for one record selection and hide scroll bars in the input form |
| * | | ⇒ | Show scroll bars in the input form (overrides second option of first optional *) |

## Description

**DISPLAY SELECTION** displays the selection of *aTable*, using the output form. The records are displayed in a scrollable list similar to that of the Design environment. If the user double-clicks a record, by default the record is displayed in the current input form. The list is displayed in the frontmost window.

To display a selection and also modify a record in the current input form after you have double-clicked on it (as you do in the Design environment window), or via the Enter in list mode, use **MODIFY SELECTION** instead of **DISPLAY SELECTION**.
All of the following information applies to both commands, except for the information on modifying records.

After **DISPLAY SELECTION** is executed, there may not be a current record. Use a command such as **FIRST RECORD** or **LAST RECORD** to select one.

The *selectMode* parameter is used to set the possibilities for selecting records in the list using the mouse. You can pass one of the following constants of the "**Form Parameters**" theme in this parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Multiple selection | Longint | 2 | The user can select several records at once. To select adjacent records, click on the first record to be selected, then press the **Shift** key before clicking on the last record you want to include in the selection. To select non-adjacent records, click on each record separately while holding down the **Ctrl** (under Windows) or **Command** (under Mac OS) key. |
| No selection | Longint | 0 | It is not be possible to select a record in the list |
| Single selection | Longint | 1 | Only one record can be selected at a time |

If you do not pass the *selectMode* parameter, the "Multiple Selection" mode is used by default.

The *enterList* parameter lets you authorize the "Enter in List" mode for the displayed list. This lets the user select and modify the record values directly in the output form. Pass **True** to enable this mode or **False** to disable it. By default, if you do not pass the *enterList* parameter, the "Enter in List" mode is disabled.
Keep in mind that with the **DISPLAY SELECTION** command, this parameter only allows the selection of the values in the list and not their modification. In fact, the **DISPLAY SELECTION** command loads the records of the current selection in Read only in the current process. Only the **MODIFY SELECTION** command allows the actual entry of values.

**Note:** The **OBJECT SET ENTERABLE** command can be used to enable or disable the Enter in list mode on the fly.

If the selection contains only one record and the first optional * is not used, the record appears in the input form instead of the output form. If the first optional * is specified, a one-record selection is displayed, using the output form. If the first optional * is specified and the user displays the record in the input form by double-clicking on it, the scroll bars will be hidden in the input form. To reverse this effect, pass the second optional *.

Custom buttons may be put in the Footer or Header area of the output form in order to terminate the execution of the **DISPLAY SELECTION** command. You can use automatic Accept or Cancel buttons to exit, or use an object method that calls **ACCEPT** or **CANCEL**. When an output form called by the **DISPLAY SELECTION** command has no buttons, only the **Escape** (Windows) or **Esc** (Mac OS) key can be used to exit the list.

During and after execution of **DISPLAY SELECTION**, the records that the user highlighted (selected) are kept in a set named *UserSet*. The *UserSet* is available within the selection display for object methods when a button is clicked or a menu item is chosen. It is also available to the project method that called **DISPLAY SELECTION** after the command was completed.

## Example 1

The following example selects all the records in the [People] table. It then uses **DISPLAY SELECTION** to display the records, and allows the user to select the records to print. Finally, it selects the records with **USE SET**, and prints them with **PRINT SELECTION**:

```
ALL RECORDS([People]) ` Select all records
DISPLAY SELECTION([People];*) ` Display the records
USE SET("UserSet") ` Use only records picked by user
PRINT SELECTION([People]) ` Print the records that the user picked
```

## Example 2

See example #6 for the **Form event** command. This example shows all the tests you may need to check in order to fully monitor the events that occur during a **DISPLAY SELECTION**.

## Example 3

To reproduce the functionality provided by, for example, the **Records** menu of the Design environment when you use **DISPLAY SELECTION** or **MODIFY SELECTION** in the Application environment, proceed as follows:

a. In the Design environment, create a menu bar with the menu commands you want, for example, Show All, Query and Order By.

b. Associate this menu bar (using the “Associated menu bar” menu in the form properties dialog box) with the output form used with **DISPLAY SELECTION** or **MODIFY SELECTION**.

c. Associate the following project methods to your menu commands:

```
 ` M_SHOW_ALL (attached to menu item Show All)
$vpCurTable:=Current form table
ALL RECORDS($vpCurTable->)

 ` M_QUERY (attached to menu item Query)
$vpCurTable:=Current form table
QUERY($vpCurTable->)

 ` M_ORDER_BY (attached to menu item Order By)
$vpCurTable:=Current form table
ORDER BY($vpCurTable->)
```

You can also use other commands, such as **PRINT SELECTION**, **QR REPORT**, and so on, to provide all the “standard” menu options you may want each time you display or modify a selection in the Application environment. Thanks to the **Current form table** command, these methods are generic, and the menu bar they support can be attached to any output form of any table.

# Displayed line number

Displayed line number -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Number of row being displayed |

## Description

The **Displayed line number** command only works with the On Display Detail form event. It returns the number of the row being processed while a list of records or list box rows is displayed on screen. If **Displayed line number** is called other than when displaying a list or a list box, it returns 0.

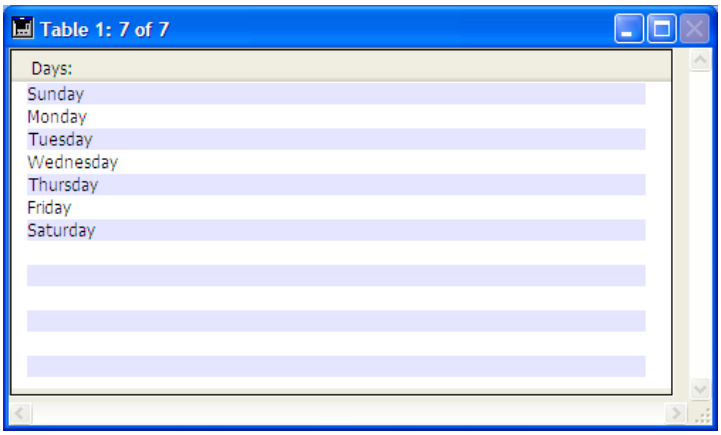
In the case of a list of records, when the displayed row is not empty (when it is linked to a record), the value returned by **Displayed line number** is identical to the value returned by **Selected record number**.

Like **Selected record number**, **Displayed line number** starts at 1. This command is useful if you want to process each row of a list form or list box displayed on screen, including empty rows.

## Example

This example lets you apply an alternating color to a list form displayed on screen, even for rows without records:

```
  `List form method
If(Form event=On Display Detail)
   If(Displayed line number% 2=0)
  `Black on white for even row text
      OBJECT SET RGB COLORS([Table 1]Field1;-1;0x00FFFFFF)
   Else
  `Black on light blue for odd row text
      OBJECT SET RGB COLORS([Table 1]Field1;-1;0x00E0E0FF)
   End if
End if
```



# End selection

End selection {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to test if record pointer is beyond the last selected record, or Default table, if omitted |
| Function result | Boolean | ⇐ | Yes (TRUE) or No (FALSE) |

## Description

**End selection** returns TRUE when the current record pointer is beyond the last record of the current selection of *aTable*. **End selection** is commonly used to check whether or not **NEXT RECORD** has moved the current record pointer past the last record. If the current selection is empty, **End selection** returns TRUE.

To move the current record pointer back into the selection, use **FIRST RECORD**, **LAST RECORD** or **GOTO SELECTED RECORD**. **PREVIOUS RECORD** does not move the pointer back into the selection.

**End selection** also returns TRUE in the last footer when a report is being printed with **PRINT SELECTION** or from the Print menu. You can use the following code to test for the last footer and print a special footer for the last page:

```
  ` Method of a form being used as output form for a summary report
$vpFormTable:=Current form table
Case of
  ` ...
  :(Form event=On Printing Footer)
  ` A footer is about to be printed
     If(End selection($vpFormTable->))
  ` Code for the last footer goes here
     Else
  ` Code for a footer goes here
     End if
End case
```

## Example

This form method is used during the printing of a report. It sets the variable *vFooter* to print in the Footer area on the last page:

```
  ` [Finances];"Summary" Form Method
Case of
  ` ...
  :(Form event=On Printing Footer)
     If(End selection([Finances]))
        vFooter:="©2001 Acme Corp." ` Set the footer for the last page
     Else
        vFooter:="" ` Clear the footer for all other pages
     End if
End case
```

# FIRST RECORD

FIRST RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to move to the first selected record, or Default table, if omitted |

## Description

**FIRST RECORD** makes the first record of the current selection of *aTable* the current record, and loads the record from disk. All query, selection, and sorting commands also set the current record to the first record. If the current selection is empty or if the current record is already the first record of the selection, **FIRST RECORD** has no effect.

This command is most often used after the **USE SET** command to begin looping through a selection of records from the first record. However, you can also call it from a subroutine if you are not sure whether or not the current record is actually the first.

## Example

The following example makes the first record of the [Customers] table the first record:

```
FIRST RECORD([Customers])
```

# GET HIGHLIGHTED RECORDS

GET HIGHLIGHTED RECORDS ( {aTable ;} setName )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table where the highlighted records will be read If omitted, table of the current form |
| setName | String | ⇒ | Set where the highlighted records will be stored |

## Description

The **GET HIGHLIGHTED RECORDS** command stores in the set designated by the *setName* parameter the highlighted records (i.e., the records highlighted by the user in the list form) in the *aTable* passed as parameter. If the *aTable* parameter is omitted, the table of the current form or subform is used.

In Design mode or when executing the **DISPLAY SELECTION** /**MODIFY SELECTION** commands, this command can be replaced by calling the UserSet system set which is automatically maintained by 4D. However, since this command allows you to pick the table that will receive highlighted records, the **GET HIGHLIGHTED RECORDS** command can also manage record selections in subforms as well. In this case, subform selections can also come from different tables. For more information about the UserSet set, refer to the **Sets** section.

The **GET HIGHLIGHTED RECORDS** command can also be called in a non-form context; however, the returned set is empty. The set designated by *setName* can be local/client, process or interprocess.

**Note:** In included subforms, the **GET HIGHLIGHTED RECORDS** command returns an empty set if the subform does not have the **Multiple** Selection Mode property. In this case, to find out the selected row, you must use the **Selected record number** command.

## Example

This method indicates how many records are selected in the subform displaying the records of the [CDs] table:

```
GET HIGHLIGHTED RECORDS([CDs];"$highlight")
ALERT(String(Records in set("$highlight"))"+"selected records.")
CLEAR SET("$highlight")
```

## System variables and sets

If the command was executed properly, the system variable OK is set to 1. Otherwise, it is set to 0.

# GOTO SELECTED RECORD

GOTO SELECTED RECORD ( {aTable ;} record )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table in which to go to the selected record, or Default table, if omitted |
| record | Longint | ⇒ | Position of record in the selection |

## Description

**GOTO SELECTED RECORD** moves to the specified record in the current selection of *aTable* and makes that record the current record. The current selection does not change. The *record* parameter is not the same as the number returned by **Record number**; it represents the record's position in the current selection. The record's position depends on how the selection is made and whether or not the selection is sorted.

**GOTO SELECTED RECORD** does nothing if:

- there are no records in the current selection
- *record* is not in the current selection,
- *record* is already the current record.

If you pass 0 in *record*, there will no longer be a current record in *aTable*. When the "single" selection mode is chosen, this allows you to deselect all the records in a list, in particular in the case of included subforms.

## Example

The following example loads data from the field [People]Last Name into the *atNames* array. An array of long integers, called *alRecNum*, is filled with numbers that will represent the selected record numbers. Both arrays are then sorted:

```
  ` Make any selection for the [People] table here
  ` ...
  ` Get the names
SELECTION TO ARRAY([People]Last Name;atNames)
  ` Create an array for the selected record numbers
$vlNbRecords:=Size of array(atNames)
ARRAY LONGINT(alRecNum;$vlNbRecords)
For($vlRecord;1;$vlNbRecords)
   alRecNum{$vlRecord}:=$vlRecord
End for
  ` Sort the arrays in alphabetical order
SORT ARRAY(atNames;alRecNum;>)
```

If the *atNames* array is displayed in a scrollable area, the user can click one of the items. Since the sorting of the two arrays is synchronized, any element in *alRecNum* provides the selected record number for the record whose name is stored in the corresponding element in *atNames*.

The following object method for *atNames* selects the correct record in the [People] selection, according to the name chosen in the scrollable area:

```
Case of
   :(Form event=On Clicked)
      If(atNames#0)
         GOTO SELECTED RECORD(alRecNum{atNames})
      End if
End case
```

# HIGHLIGHT RECORDS

HIGHLIGHT RECORDS ( {aTable }{;}{ setName {; *}} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table where records will be highlighted If omitted, table of current form |
| setName | String | ⇒ | Set of records to highlight or Userset if omitted |
| * | Operator | ⇒ | Disable the automatic scroll of the list |

## Description

The **HIGHLIGHT RECORDS** command highlights records in a list form. This operation is identical to manually selecting records in list mode by using the mouse or the **Shift+Click** or **Ctrl+Click** (Windows) or **Command+Click** (Mac OS) key combinations. The current selection is not modified.

**Note:** The set of "selected" records is updated after redrawing the records; that is, after executing the entire calling method — and not just immediately after executing **HIGHLIGHT RECORDS**.

The *aTable* parameter lets you designate the table where records will be "highlighted." This parameter can be used, in particular, to highlight the records of included subforms — which do not belong to the current table (see below).

- If you pass a valid set name to *setName*, the command is applied to the records in that set for the *table* defined.
- If you omit the *setName* parameter, the command only highlights the records in the current UserSet set. This set is only managed in Design mode and when calling the **DISPLAY SELECTION** /**MODIFY SELECTION** commands. If you want to highlight the records of a subform, you must pass a table name and set name. For more information about the UserSet set, refer to the **Sets** section.

The * parameter, when passed, disables the automatic scroll function of the list if the highlighted records are not visible. This mechanism allows customized scroll management using the **OBJECT SET SCROLL POSITION** command.

**Note:** Regarding included subforms, the **HIGHLIGHT RECORDS** command does nothing if the Selection Mode property **Multiple** is not selected for the subform. In this case, to highlight a line, you must use the **GOTO SELECTED RECORD** command.

## Example

In an output form displayed by the **MODIFY SELECTION** command, you want the user to be able to perform searches without the current selection being modified. To do this, place a **Search** button in the form and associate it with the following method:

```
SET QUERY DESTINATION(Into set;"UserSet")
QUERY
SET QUERY DESTINATION(Into current selection)
HIGHLIGHT RECORDS
```

When the user clicks the button, the standard query dialog box appears. Once the search has been validated, the records found will be highlighted without the current selection being modified.

# LAST RECORD

LAST RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to move to the last selected record, or Default table, if omitted |

## Description

**LAST RECORD** makes the last record of the current selection of *aTable* the current record and loads the record from disk. If the current selection is empty or if the current record is already the last one in the selection, **LAST RECORD** has no effect.

## Example

The following example makes the last record of the [People] table the current record:

```
LAST RECORD([People])
```

# MOBILE Return selection

MOBILE Return selection ( aTable ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table whose current selection you want to return |
| Function result | Object | ⮌ | Wakanda-compliant selection |

## Description

The **MOBILE Return selection** command returns a JSON *object* that contains the current selection of *aTable* transformed into a Wakanda-compliant entity collection.

This command is intended to be called in the context of a 4D Mobile connection, usually between your 4D application and a Wakanda application (via REST). When a 4D Mobile connection is established and appropriate access rights have been configured, a Wakanda application can execute a 4D project method that returns a value in the *$0* parameter.

The **MOBILE Return selection** command allows you to return, in *$0*, the current selection of records of the *aTable* table, in the form of an *entity collection* object in JSON format. This object is compliant with Wakanda entity collections that contain a selection of records (i.e. of *entities*).

Keep in mind that 4D Mobile accesses require specific configurations in your 4D database:

- The Web server must be launched,
- The "Activate 4D Mobile Service" option must be checked in your Database settings,
- You must have a valid license,
- Tables and fields used must have the "Expose for 4D Mobile" option checked (set by default).
- Called methods must have the "Available through 4D Mobile call" option checked (not set by default).

Note that you can pass any valid table of the database in *aTable*, and not necessarily the table with which the project method has been associated in its properties. This parameter is only used on the Wakanda side to define the objects for which the method can be called.

For more information on 4D Mobile configuration, please refer to the **4D Mobile** documentation.

## Example

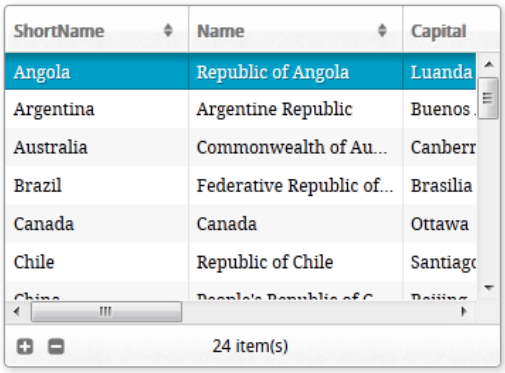
You want to display the current selection of the [Countries] table in a Wakanda grid, based on a query.

You write the following 4D method:

```
 //FindCountries project method
 //FindCountries( string ) -> object

C_TEXT($1)
C_OBJECT($0)
QUERY([Countries];[Countries]ShortName=$1+"@")
$0:=MOBILE Return selection([Countries])
```

The returned selection can be used directly in Wakanda as a valid collection.

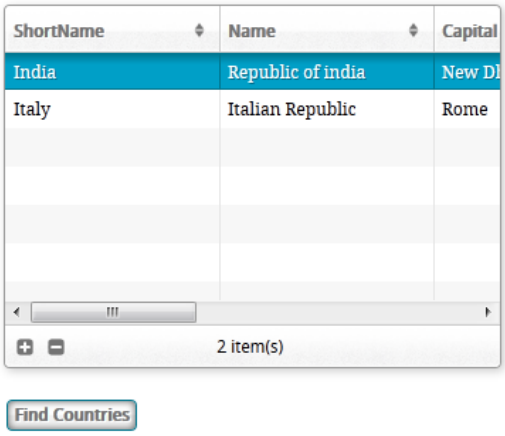
In the Wakanda server model connected to 4D via 4D Mobile, you have created a page with a grid bound to the 4D Countries table. By default, at runtime, all entities from the 4D table are displayed:




The code of the button is:

```
button1.click = function button1_click (event)      {          sources.countries.FindCountries("i", {  //we call the 4D method, "i" is passed as $1
            onSuccess:function(coll){  //callback function (asynchronous), receives $0 as parameter
sources.countries.setEntityCollection(coll.result); //replace the current entity collection                                    // with the one in the
coll.result object                  }             });            };
```

As a result, the grid is updated:



# MODIFY SELECTION

MODIFY SELECTION ( {aTable}{; selectMode}{; enterList}{; *}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to display and modify, or Default table, if omitted |
| selectMode | Longint | ⇒ | Selection mode |
| enterList | Boolean | ⇒ | Authorize Enter in list option |
| * | | ⇒ | Use output form for one record selection and hide scroll bars in the input form |
| * | | ⇒ | Show scroll bars in the input form (overrides second option of first optional *) |

## Description

**MODIFY SELECTION** does almost the same thing as **DISPLAY SELECTION**. Refer to the description of **DISPLAY SELECTION** for details. The differences between the two commands are:

1. **DISPLAY SELECTION** and **MODIFY SELECTION** enable you to display the current selected records in list mode, or in the input form when you double-click on a record. Using **MODIFY SELECTION**, you can also modify the fields of the record in the input form when you double-click on it, if it is not already in use by another process or user, or in “Enter in List” mode (if it is authorized).

2. **DISPLAY SELECTION** loads the records in Read-only mode in the current process, which means that they are not locked for writing in the other processes. **MODIFY SELECTION** places all the records of the selection in Read-Write mode, which means that they are automatically locked for writing in other processes. **MODIFY SELECTION** frees the records when its execution is completed.

# NEXT RECORD

NEXT RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to move to the next selected record, or Default table, if omitted |

## Description

**NEXT RECORD** moves the current record pointer to the next record in the current selection of *aTable* for the current process. If the current selection is empty, or if **Before selection** or **End selection** is TRUE, **NEXT RECORD** has no effect.

If **NEXT RECORD** moves the current record pointer past the end of the current selection, **End selection** returns TRUE, and there is no current record. If **End selection** returns TRUE, use **FIRST RECORD**, **LAST RECORD** or **GOTO SELECTED RECORD** to move the current record pointer back into the current selection.

## Example

See the example for **DISPLAY RECORD**.

# ONE RECORD SELECT

ONE RECORD SELECT {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table in which to reduce the selection to the current record, or Default table, if omitted |

## Description

**ONE RECORD SELECT** reduces the current selection of *aTable* to the current record. If no current record exists or if the current record is not loaded into memory (special case), **ONE RECORD SELECT** has no effect.

## Note

This command was useful to “return” a record that had been pushed and popped from the record stack back to the selection while the selection for the table was changed. **SET QUERY DESTINATION** allows you to make a query without changing the selection or the current record of a table; therefore, you no longer need to push and pop a current record in order to query its table. Consequently, **ONE RECORD SELECT** is less useful, unless you actually want to reduce the selection of a table to the current record.

# PREVIOUS RECORD

PREVIOUS RECORD {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to move to the previous selected record, or Default table, if omitted |

## Description

**PREVIOUS RECORD** moves the current record pointer to the previous record in the current selection of *aTable* for the current process. If the current selection is empty, or if **Before selection** or **End selection** is TRUE, **PREVIOUS RECORD** has no effect.

If **PREVIOUS RECORD** moves the current record pointer before the current selection, **End selection** returns TRUE, and there is no current record. If **End selection** returns TRUE, use **FIRST RECORD**, **LAST RECORD** or **GOTO SELECTED RECORD** to move the current record pointer back into the current selection.

# Records in selection

Records in selection {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return number of selected records, or Default table, if omitted |
| Function result | Longint | ⇐ | Records in selection of table |

## Description

**Records in selection** returns the number of records in the current selection of *aTable*. In contrast, **Records in table** returns the total number of records in the table.

## Example

The following example shows a loop technique commonly used to move through all the records in a selection. The same action can also be accomplished with the **APPLY TO SELECTION** command:

```
FIRST RECORD([People]) ` Start at first record in the selection
For($vlRecord;1;Records in selection([People])) ` Loop once for each record
   Do Something ` Do something with the record
   NEXT RECORD([People]) ` Move to the next record
End for
```

# REDUCE SELECTION

REDUCE SELECTION ( {aTable ;} number )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to reduce the selection, or Default table, if omitted |
| number | Longint | ⇒ | Number of records to keep selected |

## Description

**REDUCE SELECTION** creates a new selection of records for *aTable*. The command returns the first *number* of records from the current selection *table*. **REDUCE SELECTION** is applied to the current selection of *aTable* in the current process. It changes the current selection of *aTable* for the current process; the first record of the new selection is the current record.

**Note:** If the statement **REDUCE SELECTION***(aTable;0)* is executed, there is no longer any selection nor any current records in the table.

## Example

The following example first finds the correct statistics for a worldwide contest among the dealers in over 20 countries. For each country, the 3 best dealers who have sold product worth more than $50,000 and who are among the 100 best dealers in the world are awarded a prize. With a few lines of code, this complex request can be executed by using indexed searches:

```
CREATE EMPTY SET([Dealers];"Winners") ` Create an empty set
SCAN INDEX([Dealers]Sales amount;100;<) ` Scan from the end of the index
CREATE SET([Dealers];"100 best Dealers") ` Put the selected records in a set
For($Country;1;Records in table([Countries])) ` For each Country
 ` Search for the dealers in this country
   QUERY([Dealers];[Dealers]Country=[Countries]Name;*) ` ...who sold for more than $50,000
   QUERY(&;[Dealers];[Dealers]Sales amount>=50000)
   CREATE SET([Dealers];"WinnerDealers") ` Put them in a set
 ` They should be in the group of 100 best dealers
   INTERSECTION("WinnerDealers";"100 best Dealers";"WinnerDealers")
   USE SET("WinnerDealers") ` Potential winners for the country
 ` Sort them by the results in descending order
   ORDER BY([Dealers];[Dealers]Sales amount;<)
   REDUCE SELECTION([Dealers];3) ` Take the 3 best Dealers
   CREATE SET([Dealers];"WinnerDealers") ` The winners for the country
 ` Put them in the worldwide winners list
   UNION("WinnerDealers";"TheWinners";"TheWinners")
End for
CLEAR SET("100 best Dealers") ` Don't need this set anymore
CLEAR SET("WinnerDealers") ` Don't need this set anymore
USE SET("The Winners") ` Here you have the Winners
CLEAR SET("The Winners") ` Don't need this set anymore
OUTPUT FORM([Dealers];"Prize letter") ` Select the letter
PRINT SELECTION([Dealers]) ` Print the letters
```

# SCAN INDEX

SCAN INDEX ( aField ; number {; > or <} )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ➡ | Indexed field on which to scan index |
| number | Longint | ➡ | Number of records to return |
| > or < | Operator | ➡ | > from beginning of index < from end of index |

## Description

**SCAN INDEX** returns a selection of *number* records from the table containing the *aField* field. If you pass <, **SCAN INDEX** returns the *number* of records from the end of the index (high values). If you pass >, **SCAN INDEX** returns the *number* of records from the beginning of the index (low values). This command is very efficient because it uses the index to perform the operation.

**Note:** The selection obtained is not sorted.

**SCAN INDEX** only works on indexed fields. This command changes the current selection of the table for the current process and loads the first record of the selection as the current record.

If you specify more records than exist in the table, **SCAN INDEX** will return all the records.

**Note:** This command does not support Object type fields.

## Example

The following example mails letters to 50 of the worst customers and then to 50 of the best customers:

```
SCAN INDEX([Customers]TotalDue;50;<) ` Get the 50 worst customers
ORDER BY([Customers]Zipcode;>) ` Sort by Zip codes
FORM SET OUTPUT([Customers];"ThreateningMail")
PRINT SELECTION([Customers]) ` Print the letters
SCAN INDEX([Customers]TotalDue;50;>) ` Get the 50 best customers
ORDER BY([Customers]Zipcode;>) ` Sort by Zip codes
FORM SET OUTPUT([Customers];"Thanks Letter")
PRINT SELECTION([Customers]) ` Print the letters
```

# Selected record number

Selected record number {( aTable )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to return the selected record number or Default table, if omitted |
| Function result | Longint | ⇐ | Selected record number of current record |

## Description

**Selected record number** returns the position of the current record within the current selection of *aTable*.

If the selection is not empty and if the current record is within the selection, **Selected record number** returns a value between *1* and **Records in selection**. If the selection is empty, of if there is no current record, it returns *0* (zero).

The selected record number is not the same as the number returned by **Record number**, which returns the physical record number in the table. The selected record number depends on the current selection and the current record.

## Example

The following example saves the current selected record number in a variable:

```
CurSelRecNum:=Selected record number([People]) ` Get the selected record number
```

# TRUNCATE TABLE

TRUNCATE TABLE {( aTable )}

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table where all records will be deleted or Default table if this parameter is omitted |

## Description

The **TRUNCATE TABLE** command quickly deletes all the records of *aTable*. After calling the command, there is no longer any current selection or current record.

The effect of this command is similar to that of an **ALL RECORDS** / **DELETE SELECTION** sequence; however, its functioning differs on the following points:

- No trigger is called
- The referential integrity of the data is not checked.
- No transaction must be underway in the process executing **TRUNCATE TABLE**. If this if the case, the command does nothing and the OK system variable is set to 0
- If one or more records are locked by another process, the command fails: an error is generated and the OK system variable is set to 0. The LockedSet system set is not created.
- If *aTable* is already empty, **TRUNCATE TABLE** does nothing and sets the OK variable to 1.
- If *aTable* is in read-only, **TRUNCATE TABLE** does nothing and sets the OK variable to 0.
- The operation is recorded in the log file if there is one.

The **TRUNCATE TABLE** command should therefore be used with caution but is very effective in certain cases, for example, such as quickly deleting temporary data

**Note:** The concept and functioning of this command is similar to that of the SQL TRUNCATE (TABLE) command.

## System variables and sets

If the command has been executed correctly, the OK system variable is set to 1. Otherwise, it is set to 0.

# Sets

- Sets
- ADD TO SET
- CLEAR SET
- COPY SET
- CREATE EMPTY SET
- CREATE SET
- CREATE SET FROM ARRAY
- DIFFERENCE
- INTERSECTION
- Is in set
- LOAD SET
- Records in set
- REMOVE FROM SET
- SAVE SET
- UNION
- USE SET

# Sets

Sets offer you a powerful, swift means for manipulating record selections. Besides the ability to create sets, relate them to the current selection, and store, load, and clear sets, 4D offers three standard set operations:

- Intersection
- Union
- Difference.

## Sets and the Current Selection

A set is a compact representation of a selection of records. The idea of sets is closely bound to the idea of the current selection. Sets are generally used for the following purposes:

- To save and later restore a selection when the order does not matter
- To access the selection a user made on screen (the *UserSet*)
- To perform a logical operation between selections.

The current selection is a list of references that points to each record that is currently selected. The list exists in memory. Only currently selected records are in the list. A selection doesn't actually contain the records, but only a list of references to the records. Each reference to a record takes 4 bytes in memory. When you work on a table, you always work with the records in the current selection. When a selection is sorted, only the list of references is rearranged. There is only one current selection for each table inside a process.

Like a current selection, a set represents a selection of records. A set does this by using a very compact representation for each record. Each record is represented by one bit (one-eighth of a byte). Operations using sets are very fast, because computers can perform operations on bits very quickly. A set contains one bit for every record in the table, whether or not the record is included in the set. In fact, each bit is equal to 1 or 0, depending on whether or not the record is in the set.

Sets are very economical in terms of RAM space. The size of a set, in bytes, is always equal to the total number of records in the table divided by 8. For example, if you create a set for a table containing 10,000 records, the set takes up 1,250 bytes, which is about 1.2K in RAM.

There can be many sets for each table. In fact, sets can be saved to disk separately from the database. To change a record belonging to a set, first you must use the set as the current selection, then modify the record or records.

A set is never in a sorted order—the records are simply indicated as belonging to the set or not. On the other hand, a named selection is in sorted order, but it requires more memory in most cases. For more information about named selections, see the **Named Selections** section.

A set "remembers" which record was the current record at the time the set was created. The following table compares the concepts of the current selection and of sets:

| Comparison | Current Selection | Sets |
|---|---|---|
| Number per table | 1 | 0 to many |
| Sortable | Yes | No |
| Can be saved on disk | No | Yes |
| RAM per record(in bytes) | Number of selected records * 4 | Total number of records/8 |
| Combinable | No | Yes |
| Contains current record | Yes | Yes, as of the time the set was created |

When you create a set, it belongs to the table from which you created it. Set operations can be performed only between sets belonging to the same table.

Sets are independent from the data. This means that after changes are made to a file, a set may no longer be accurate. There are many operations that can cause a set to be inaccurate. For example, if you create a set of all the people from New York City, and then change the data in one of those records to "Boston" the set would not change, because the set is just a representation of a selection of records. Deleting records and replacing them with new ones also changes a set, as well as compacting the data. Sets can be guaranteed to be accurate only as long as the data in the original selection has not been changed.

## Process and Interprocess Sets

You can have the following three types of sets:

- **Process sets**: A process set can only be accessed by the process in which it has been created. **LockedSet** is a process set. Process sets are cleared as soon as the process method ends. Process sets do not need any special prefix in the name.
- **Interprocess sets**: A set is an interprocess set if the name of the set is preceded by the symbols (<>) — a "less than" sign followed by a "greater than" sign. **Note**: This syntax can be used on both Windows and Macintosh. Also, on Macintosh only, you can use the diamond (Option-Shift-V on a US keyboard).
  An interprocess set is "visible" to all the processes of the database.
  In client/server mode, an interprocess set is "visible" to processes of the machine where it was created (client or server).
  The name of an interprocess set must be unique in the database.
- **Local Sets/Client Sets**: Local/client sets are intended for use in client/server mode. The name of a local/client set is always preceded by the dollar sign ($) -- except for the **UserSet** system set. Unlike other types of sets, a local/client set is stored on the client machine.

**Notes:**

- The maximum size of a set name is 255 characters (excluding <> and $ symbols).
- For more information about the use of sets in client/server mode, please refer to **4D Server, Sets and Named Selections** of the 4D Server Reference Manual.

## Visibility of Sets

The following table indicates the principles concerning the visibility of sets depending on their scope and where they were created:

| | Client Process | Other client processes | Other clients | Server process | Other server processes |
|---|---|---|---|---|---|
| **Creation in a client process** | | | | | |
| $test | x | | | | |
| test | x | | | x (Trigger) | |
| <>test | x | x | | | |
| **Creation in a server process** | | | | | |
| $test | | | | x | |
| test | | | | x | |
| <>test | | | | x | x |

## Sets and Transactions

A set can be created inside a transaction. It is possible to create a set of the records created inside a transaction and a set of records created or modified outside of a transaction. When the transaction ends, the set created during the transaction should be cleared, because it may not be an accurate representation of the records, especially if the transaction was canceled.

## Set Example

The following example deletes duplicate records from a table which contains information about people. A *For...End for* loop moves through all the records, comparing the current record to the previous record. If the name, address, and zip code are the same, then the record is added to a set. At the end of the loop, the set is made the current selection and the (old) current selection is deleted:

```
CREATE EMPTY SET([People];"Duplicates")
  ` Create an empty set for duplicate records
ALL RECORDS([People])
  ` Select all records
  ` Sort the records by ZIP, address, and name so
  ` that the duplicates will be next to each other
ORDER BY([People];[People]ZIP;>;[People]Address;>;[People]Name;>)
  ` Initialize variables that hold the fields from the previous record
$Name:=[People]Name
$Address:=[People]Address
```

```
$ZIP:=[People]ZIP
 ` Go to second record to compare with first
NEXT RECORD([People])
For($i;2;Records in table([People]))
 ` Loop through records starting at 2
 ` If the name, address, and ZIP are the same as the
 ` previous record then it is a duplicate record.
   If(([People]Name=$Name) & ([People]Address=$Address) & ([People]ZIP=$ZIP))
 ` Add current record (the duplicate) to set
      ADD TO SET([People];"Duplicates")
   Else
 ` Save this record's name, address, and ZIP for comparison with the next record
      $Name:=[People]Name
      $Address:=[People]Address
      $ZIP:=[People]ZIP
   End if
 ` Move to the next record
   NEXT RECORD([People])
End for
 ` Use duplicate records that were found
USE SET("Duplicates")
 ` Delete the duplicate records
DELETE SELECTION([People])
 ` Remove the set from memory
CLEAR SET("Duplicates")
```

As an alternative to immediately deleting records at the end of the method, you could display them on screen or print them, so that a more detailed comparison can be made.

## The UserSet System Set

4D maintains a system set named *UserSet*, which automatically stores the most recent selection of records highlighted on screen by the user. Thus, you can display a group of records with **MODIFY SELECTION** or **DISPLAY SELECTION**, ask the user to select from among them and turn the results of that manual selection into a selection or into a set that you name.

**4D Server**: Although its name does not begin with the character "$", the *UserSet* system set is a client set. So, when using **INTERSECTION**, **UNION** and **DIFFERENCE**, make sure you compare *UserSet* only to client sets. For more information, please refer to the descriptions of these commands as well as to the **4D Server, Sets and Named Selections** section of the 4D Server Reference Manual.

There is only one *UserSet* for a process. Each table does not have its own *UserSet*. *UserSet* becomes “owned” by a table when a selection of records is displayed for the table.

4D manages the *UserSet* set for list forms displayed in Design mode or using the **MODIFY SELECTION** or **DISPLAY SELECTION** commands. However, this mechanism is not active for subforms.

The following method illustrates how you can display records, allow the user to select some of them, and then use *UserSet* to display the selected records:

```
 ` Display all records and allow user to select any number of them.
 ` Then display this selection by using UserSet to change the current selection.
FORM SET OUTPUT([People];"Display") ` Set the output layout
ALL RECORDS([People]) ` Select all people
ALERT("Press Ctrl or Command and Click to select the people required.")
DISPLAY SELECTION([People]) ` Display the people
USE SET("UserSet") ` Use the people that were selected
ALERT("You chose the following people.")
DISPLAY SELECTION([People]) ` Display the selected people
```

## The LockedSet System Set

The **APPLY TO SELECTION**, **DELETE SELECTION**, **ARRAY TO SELECTION** and **JSON TO SELECTION** commands create a set named **LockedSet** when used in a multi-processing environment.
Query commands also create a *LockedSet* system set when they find locked records in the 'query and lock' context (see the **SET QUERY AND LOCK** command).
*LockedSet* indicates which records were locked during the execution of the command.

# ADD TO SET

ADD TO SET ( {aTable ;} set )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Current record's table, or Default table, if omitted |
| set | String | ⇒ | Name of the set to which to add the current record |

## Description

**ADD TO SET** adds the current record of *aTable* to *set*. The set must already exist; if it does not, an error occurs. If a current record does not exist for *aTable*, **ADD TO SET** has no effect.

# CLEAR SET

CLEAR SET ( set )

| Parameter | Type | | Description |
|---|---|---|---|
| set | String | ⇒ | Name of the set to clear from memory |

## Description

**CLEAR SET** clears *set* from memory and frees the memory used by *set*. The command does not affect tables, selections, or records. To save a set before clearing it, use the **SAVE SET** command. Since sets use memory, it is good practice to clear them when they are no longer needed.

## Example

See the example for **USE SET**.

# COPY SET

COPY SET ( srcSet ; dstSet )

| Parameter | Type | | Description |
|---|---|---|---|
| srcSet | String | ⇨ | Source set name |
| dstSet | String | ⇨ | Destination set name |

## Description

The **COPY SET** command copies the contents of the set *srcSet* into the set *dstSet*.

Each of these sets can be of the process, interprocess or local/client type. The two sets do not have to be the same type (as shown in the examples below), so long as they are both visible on the machine. For more information about this point, refer to "**Visibility of Sets**".

## Example 1

The following example, in Client/Server, copies the local set "*$SetA",* maintained on the client machine, to the process set *"SetB",* maintained on the server machine:

```
COPY SET("$SetA";"SetB")
```

## Example 2

The following example, in Client/Server, copies the process set "*SetA*", maintained on the server machine, to the local process set *"$SetB",* maintained on the client machine:

```
COPY SET("SetA";"$SetB")
```

# CREATE EMPTY SET

CREATE EMPTY SET ( {aTable ;} set )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to create an empty set, or Default table, if omitted |
| set | String | ⇒ | Name of the new empty set |

## Description

**CREATE EMPTY SET** creates a new empty set, *set*, for *aTable*. You can add records to this set with the **ADD TO SET** command. If a set with the same name already exists, the existing set is cleared by the new set.

**Note:** You do not need to use **CREATE EMPTY SET** before using **CREATE SET**.

## Example

Please refer to the examples of the **Sets** section.

# CREATE SET

CREATE SET ( {aTable ;} set )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to create a set from the selection, or Default table, if omitted |
| set | String | ⇒ | Name of the new set |

## Description

**CREATE SET** creates a new set, *set*, for *aTable*, and places the current selection in *set*. The current record pointer for the table is saved with *set*. If *set* is used with **USE SET**, the current selection and current record are restored. As with all sets, there is no sorted order; when *set* is used, the default order is used. If a set with the same name already exists, the existing set is cleared by the new set.

## Example

The following example creates a set after doing a search, in order to save the set to disk:

```
QUERY([People]) ` Let the user do a search
CREATE SET([People];"SearchSet") ` Create a new set
SAVE SET("SearchSet";"MySearch") ` Save the set on disk
```

# CREATE SET FROM ARRAY

CREATE SET FROM ARRAY ( aTable ; recordsArray {; setName} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table of the set |
| recordsArray | Longint, Boolean array | ⇒ | Array of record numbers, or Array of booleans (True = the record is in the set, False = the record is not in the set) |
| setName | String | ⇒ | Name of the set to create, or Apply the command to the Userset if omitted |

## Description

The **CREATE SET FROM ARRAY** command creates *setName* from:

- Either an array of absolute record numbers *recordsArray* from *aTable*,
- Or an array of booleans *recordsArray*. In this case, the values of the array indicate if each record in the table belongs (**True**) or not (**False**) to *setName*.

When you use this command and pass a Longint array in *recordsArray*, all the numbers in the array represent the list of record numbers that are in *setName*. If a number is invalid (for example, if a record has not been created), the error -10503 is generated.

When you use this command and pass a Boolean array in *recordsArray*, the Nth element of the array indicates whether the "Nth" record is contained (**True**) or not (**False**) in *setName*. Usually, the number of elements in the array must equal the number of records in the table. If the array is smaller than the number of records, only the records defined by the array will be in the set.

**Note:** With a Boolean array, this command uses the elements from 0 to N-1.

If you do not pass the *setName* parameter or if you pass an empty string, the command will be applied to the *Userset* system set.

## Error management

In a Longint array, if a record number is invalid (record not created), the error -10503 is generated.

# DIFFERENCE

DIFFERENCE ( set1 ; set2 ; resultSet )

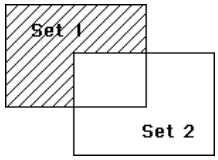
| Parameter | Type | | Description |
|---|---|---|---|
| set1 | String | ⇒ | Set |
| set2 | String | ⇒ | Set to subtract |
| resultSet | String | ⇒ | Resulting set |

## Description

**DIFFERENCE** compares *set1* and *set2* and excludes all records that are in *set2* from the *resultSet*. In other words, a record is included in the *resultSet* only if it is in *set1*, but not in *set2*. The following table shows all possible results of a set Difference operation.

| Set1 | Set2 | Result Set |
|---|---|---|
| Yes | No | Yes |
| Yes | Yes | No |
| No | Yes | No |
| No | No | No |

The result of a Difference operation is depicted here. The shaded area is the result set.




The *resultSet* is created by **DIFFERENCE**. The *resultSet* replaces any existing set having the same name, including *set1* and *set2*. Both *set1* and *set2* must be from the same table. The *resultSet* belongs to the same table as *set1* and *set2*.

**4D Server:** In Client/Server mode, sets are "visible" depending on their type (interprocess, process and local) and where they were created (server or client). **DIFFERENCE** requires all three sets to be visible on the same machine. See the paragraph **4D Server, Sets and Named Selections** in the 4D Server Reference manual for more information.

## Example

This example excludes the records that a user selects from a displayed selection. The records are displayed on screen with the following line:

```
DISPLAY SELECTION([Customers]) //Display the customers in a list
```

At the bottom of the list of records is a button with an object method. The object method excludes the records that the user has selected (the set named “UserSet”), and displays the reduced selection:

```
CREATE SET([Customers];"$Current") //Create a set of current selection
DIFFERENCE("$Current";"UserSet";"$Current") //Exclude selected records
USE SET("$Current") //Use the new set
CLEAR SET("$Current") //Clear the set
```

# INTERSECTION

INTERSECTION ( set1 ; set2 ; resultSet )

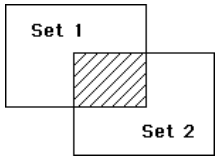
| Parameter | Type | | Description |
|---|---|---|---|
| set1 | String | ⇒ | First set |
| set2 | String | ⇒ | Second set |
| resultSet | String | ⇒ | Resulting set |

## Description

**INTERSECTION** compares *set1* and *set2* and selects only the records that are in both. The following table lists all possible results of a set Intersection operation.

| Set1 | Set2 | Result Set |
|---|---|---|
| Yes | No | No |
| Yes | Yes | Yes |
| No | Yes | No |
| No | No | No |

The graphical result of an Intersection operation is displayed here. The shaded area is the result set.




The *resultSet* is created by **INTERSECTION**. The *resultSet* replaces any existing set having the same name, including *set1* and *set2*. Both *set1* and *set2* must be from the same table. The *resultSet* belongs to the same table as *set1* and *set2*. If the same current record is set in both *set1* and *set2*, it remains memorized in the *resultSet*. Otherwise, *resultSet* does not have a current record.

**4D Server:** In Client/Server mode, sets are "visible" depending on their type (interprocess, process and local) and where they were created (server or client). **INTERSECTION** requires all three sets to be visible on the same machine. See the **4D Server, Sets and Named Selections** section in the 4D Server Reference manual for more information.

## Example

The following example finds the customers who are served by two sales representatives, Joe and Abby. Each sales representative has a set that represents his or her customers. The customers that are in both sets are represented by both Joe and Abby:

```
INTERSECTION("Joe";"Abby";"Both") ` Put customers in both sets in Both
USE SET("Both") ` Use the set
CLEAR SET("Both") ` Clear this set but save the others
DISPLAY SELECTION([Customers]) ` Display customers served by both
```

# Is in set

Is in set ( set ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| set | String | ⇒ | Name of the set to test |
| Function result | Boolean | ⇐ | Current record of set's table is in set (True) or Current record of set's table is not in set (False) |

## Description

**Is in set** tests whether or not the current record for the table is in *set*. The **Is in set** function returns TRUE if the current record of the table is in *set*, and returns FALSE if the current record of the table is not in *set*.

## Example

The following example is a button object method. It tests to see whether or not the currently displayed record is in the set of best customers:

```
If(Is in set("Best")) ` Check if it is a good customer
   ALERT("They are one of our best customers.")
Else
   ALERT("They are not one of our best customers.")
End if
```

# LOAD SET

LOAD SET ( {aTable ;} set ; document )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to which the set belongs, or Default table, if omitted |
| set | String | ⇒ | Name of the set to be created in memory |
| document | String | ⇒ | Document holding the set |

## Description

**LOAD SET** loads a set from *document* that was saved with the **SAVE SET** command.

The set that is stored in *document* must be from *aTable*. The set created in memory is overwritten if it already exists.

The *document* parameter is the name of the disk document containing the set. The document need not have the same name as the set. If you supply an empty string for *document*, an Open File dialog box appears so that the user can choose the set to load.

Remember that a set is a representation of a selection of records at the moment that the set is created. If the records represented by the set change, the set may no longer be accurate. Therefore, a set loaded from disk should represent a group of records that does not change frequently. A number of things can make a set invalid: modifying a record of the set, deleting a record of the set, or changing the criteria that determined a set.

## Example

The following example uses **LOAD SET** to load a set of the Acme locations in New York:

```
LOAD SET([Companies];"NY Acme";"NYAcmeSt") ` Load the set into memory
USE SET("NY Acme") ` Change current selection to NY Acme
CLEAR SET("NY Acme") ` Clear the set from memory
```

## System variables and sets

If the user clicks Cancel in the Open File dialog box, or there is an error during the load operation, the OK system variable is set to 0. Otherwise, it is set to 1.

# Records in set

Records in set ( set ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| set | String | ⇒ | Name of the set to test |
| Function result | Longint | ⇐ | Number of records in set |

## Description

**Records in set** returns the number of records in *set*. If *set* does not exist, or if there are no records in *set*, the command returns 0.

## Example

The following example displays an alert telling what percentage of the customers are rated as the best:

```
  ` First calculate the percentage
$Percent :=(Records in set("Best")/Records in table([Customers]))*100
  ` Display an alert with the percentage
ALERT(String($Percent;"##0%")+" of our customers are the best.")
```

# REMOVE FROM SET

REMOVE FROM SET ( {aTable ;} set )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Current record's table, or Default table, if omitted |
| set | String | ⇒ | Name of the set from which to remove the current record |

## Description

**REMOVE FROM SET** removes the current record of *aTable* from *set*. The set must already exist; if it does not, an error occurs. If a current record does not exist for *aTable*, the command has no effect.

# SAVE SET

SAVE SET ( set ; document )

| Parameter | Type | | Description |
|---|---|---|---|
| set | String | ⇒ | Name of the set to save |
| document | String | ⇒ | Name of the disk file to which to save the set |

## Description

**SAVE SET** saves *set* to *document*, a document on disk.

The *document* does not need to have the same name as the set. If you supply an empty string for *document*, a Create File dialog box appears so that the user can enter the name of the document. You can load a saved set with the **LOAD SET** command.

If the user clicks Cancel in the Save File dialog box, or if there is an error during the save operation, the OK system variable is set to 0. Otherwise, it is set to 1.

**SAVE SET** is often used to save to disk the results of a time-consuming search.

**WARNING:** Remember that a set is a representation of a selection of records at the moment that the set is created. If the records represented by the set change, the set may no longer be accurate. Therefore, a set saved to disk should represent a group of records that does not change frequently. A number of things can make a set invalid: modifying a record of the set, deleting a record of the set, or changing the criteria that determined the set. Also remember that sets do not save field values.

## Example

The following example displays the Save File dialog box, which the user can enter the name of the document that contains the set:

```
SAVE SET("SomeSet";"")
```

## System variables and sets

If the user clicks Cancel in the Save File dialog box, or if there is an error during the load operation, the OK system variable is set to 0. Otherwise, it is set to 1.

# UNION

UNION ( set1 ; set2 ; resultSet )

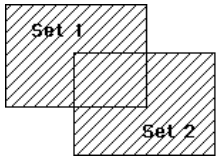
| Parameter | Type | | Description |
|---|---|---|---|
| set1 | String | ➡ | First set |
| set2 | String | ➡ | Second set |
| resultSet | String | ➡ | Resulting set |

## Description

**UNION** creates a set that contains all records from *set1* and *set2*. The following table shows all possible results of a set Union operation.

| Set1 | Set2 | Result Set |
|---|---|---|
| Yes | No | Yes |
| Yes | Yes | Yes |
| No | Yes | Yes |
| No | No | No |

The result of a Union operation is depicted here. The shaded area is the result set.




The *resultSet* is created by **UNION**. The *resultSet* replaces any existing set having the same name, including *set1* and *set2*. Both *set1* and *set2* must be from the same table. The *resultSet* belongs to the same table as *set1* and *set2*. The current record for the *resultSet* is the current record from *Set1*.

**4D Server:** In Client/Server mode, sets are "visible" depending on their type (interprocess, process and local) and where they were created (server or client). **UNION** requires that all three sets be visible on the same machine. See the paragraph **4D Server, Sets and Named Selections** in the 4D Server Reference manual for more information.

## Example

This example adds records to a set of best customers. The records are displayed on screen with the first line. After the records are displayed, a set of the best customers is loaded from disk, and any records that the user selected (the set named “UserSet”) are added to the set. Finally, the new set is saved on disk:

```
ALL RECORDS([Customers]) ` Select all the customers
DISPLAY SELECTION([Customers]) ` Display all the customers in a list
LOAD SET("$Best";"$SaveBest") ` Load the set of best customers
UNION("$Best";"UserSet";"$Best") ` Add any selected to the set
SAVE SET("$Best";"$SaveBest") ` Save the set of best customers
```

# USE SET

USE SET ( set )

| Parameter | Type | | Description |
|---|---|---|---|
| set | String | ➡ | Name of the set to use |

## Description

**USE SET** makes the records in *set* the current selection for the table to which the set belongs.

When you create a set, the current record is "remembered" by the set. **USE SET** retrieves the position of this record and makes it the new current record. If you delete this record before you execute **USE SET**, 4D selects the first record in the set as the current record. The set commands **UNION**, **INTERSECTION**, **DIFFERENCE** and **ADD TO SET** reset the current record. Also, if you create a set that does not contain the position of the current record, **USE SET** selects the first record in the set as the current record.

**WARNING:** Remember that a set is a representation of a selection of records at the moment that the set is created. If the records represented by the set do change, the set may no longer be accurate. Therefore, a set saved to disk should represent a group of records that does not change frequently. A number of things can invalidate a set invalid: modifying a record of the set, deleting a record of the set, or changing the criteria that determined the set.

## Example

The following example uses **LOAD SET** to load a set of the Acme locations in New York. It then uses **USE SET** to make the loaded set the current selection:

```
LOAD SET([Companies];"NY Acme";"NYAcmeSt") ` Load the set into memory
USE SET("NY Acme") ` Change current selection to NY Acme
CLEAR SET("NY Acme") ` Clear the set from memory
```

# Spell Checker

- Support of Hunspell dictionaries
- SPELL ADD TO USER DICTIONARY
- SPELL CHECK TEXT
- SPELL CHECKING
- SPELL Get current dictionary
- SPELL GET DICTIONARY LIST
- SPELL SET CURRENT DICTIONARY

## Support of Hunspell dictionaries

---

Beginning with versions 12.4 and 13, 4D supports OpenSource "Hunspell" dictionaries. For more information about these dictionaries, refer to the following site: http://hunspell.sourceforge.net/

4D uses the same format as MySpell and OpenOffice 2.x: an .aff file and a .dic file with the same name. For example, the "fr-modern" dictionary consists of the *fr-modern.aff* and *fr-modern.dic* files.

To be able to use a Hunspell dictionary in your 4D application, you must install its .aff and .dic files at one of the following locations:

- in the 4D application: <4D>/Resources/Spellcheck/Hunspell/
- in the 4D database: <Database_Files>/Resources/Hunspell

Both locations are compatible: the database folder is parsed first, then it is completed by the one in the 4D application, which means that you can encapsulate specialized dictionaries with your 4D databases. If two dictionaries with the same name are present in both locations, it is the one in the database that is taken into account.

You can download Hunspell dictionaries here: http://wiki.services.openoffice.org/wiki/Dictionaries

Users can also add existing dictionaries or create new ones. The process for creating these dictionaries is the same as that for Cordial user dictionaries (see **Appendix D: Use of specialized dictionaries** of the *Design Reference* manual). User dictionaries are stored in UTF-8 format.

# SPELL ADD TO USER DICTIONARY

SPELL ADD TO USER DICTIONARY ( words )

| Parameter | Type | | Description |
|---|---|---|---|
| words | Text, Text array | ⇒ | Word or list of words to add to user dictionary |

## Description

The **SPELL ADD TO USER DICTIONARY** command adds one or more words to the current user dictionary.

A user dictionary is a dictionary containing words added by the user to the current dictionary. This dictionary is a file named *UserDictionaryxxx.dic* (where xxx represents the ID of the current dictionary) that is created automatically in the current 4D folder. There is a user dictionary for each current dictionary used.

In *words*, pass a text string or a text array containing the word(s) to be added to the user dictionary. If one of the words is already found in the dictionary, it is ignored by the command.

## Example

Adding proper nouns to the user dictionary:

```
ARRAY TEXT($arrTwords;0)
APPEND TO ARRAY($arrTwords;"4D")
APPEND TO ARRAY($arrTwords;"Wakanda")
APPEND TO ARRAY($arrTwords;"Clichy")
SPELL ADD TO USER DICTIONARY($arrTwords)
```

# SPELL CHECK TEXT

SPELL CHECK TEXT ( text ; errPos ; errLength ; checkPos ; arrSuggest )

| Parameter | Type | | Description |
|---|---|---|---|
| text | Text | ➡ | Text to check |
| errPos | Longint | ⬅ | Position of first character of unknown word |
| errLength | Longint | ⬅ | Length of unknown word |
| checkPos | Longint | ➡ | Start position for check |
| arrSuggest | Text array | ⬅ | List of suggestions |

## Description

The **SPELL CHECK TEXT** command checks the contents of the *text* parameter beginning from the *checkPos* character and returns the position of the first unknown word it finds (if any).

This command returns the position of the first character of this unknown word in *errPos* and its length in *errLength*. The *arrSuggest* array receives the correction suggestion(s) proposed by the spell checker.

If the check starts without error and an unknown word is found, the OK system variable is set to 0. If an initialization error occurs during the check, or if no unknown words are found, OK is set to 1.

**Note OS X:** Under OS X, when the native spell checker is enabled, this command does not support grammar correction.

## Example

We want to count the number of possible errors in a text:

```
$pos:=1
$errCount:=0
ARRAY TEXT($tErrors;0)
ARRAY TEXT($tSuggestions;0)
Repeat
   SPELL CHECK TEXT($myText;$errPos;$errLength;$pos;$tSuggestions)
   If(OK=0)
      $errCount:=$errCount+1 // count any errors
      $errorWord:=Substring($myText;$errPos;$errLength)
      APPEND TO ARRAY($errors;$errorWord) // array of errors
      $pos:=$errPos+$errLength  //continue check
   End if
Until(OK=1)
 // In the end $errCount=Size of array($errorWord)
```

# SPELL CHECKING

SPELL CHECKING
Does not require any parameters

## Description

The **SPELL CHECKING** command triggers the spell check of the field or variable having the focus in the currently displayed form. The object checked must be of the string or text type.

**Note:** If you want to trigger the spellcheck by clicking a button in the form, make sure that this button does not have the "Focusable" property set.

Spell checking starts with the first word of the field or variable. If an unknown word is detected, the spell check dialog box appears (for more information, refer to the Design Reference manual of 4D). 4D uses the current dictionary (corresponding to the language of the application) unless you have used the **SPELL SET CURRENT DICTIONARY** command.

**Warning:** The **SPELL CHECKING** command affects the text that is being entered in the form, and not the associated data source (field or variable). This means that if you call this command from the On Data Change or On Losing Focus form events (not recommended), it will not affect the stored text since 4D has already assigned the entered text to the data source at this point. In this case, you need to assign the edited result to the data source yourself, using the **Get edited text** command. For example:

```
If(Form event=On Data Change)
   SPELL CHECKING
   theVariable:=Get edited text
End if
```

# SPELL Get current dictionary

SPELL Get current dictionary -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | ID of dictionary used for spell check |

## Description

The **SPELL Get current dictionary** command returns the ID number of the dictionary being used.

## Example

We want to display the language of the current dictionary:

```
 // List of loaded dictionaries
SPELL GET DICTIONARY LIST($IDs_al;$Codes_at;$Names_at)
$curLangCode:=SPELL Get current dictionary
$countryName:=$Names_at{Find in array($IDs_al;$curLangCode)}
 // Display message
ALERT("Current dictionary: "+$countryName) // Spanish
```

# SPELL GET DICTIONARY LIST

SPELL GET DICTIONARY LIST ( langID ; langFiles ; langNames )

| Parameter | Type | | Description |
|---|---|---|---|
| langID | Longint array | ⇦ | Unique ID of languages |
| langFiles | Text array | ⇦ | Names of language files installed |
| langNames | Text array | ⇦ | Local names of languages |

## Description

The **SPELL GET DICTIONARY LIST** command returns, in the *langID*, *langFiles* and *langNames* arrays, the IDs, file names and language names corresponding to the Hunspell dictionary files installed on the machine.

**Note:** For more information about Hunspell dictionaries, refer to the **Support of Hunspell dictionaries** section.

- *langID* receives the ID numbers generated automatically and used with the **SPELL SET CURRENT DICTIONARY** command.
  Note that the IDs are unique and based on the file names. This command is mainly useful during development; you do not have to regenerate the IDs each time the database is executed.
- *langFiles* receives the names of the dictionary files (without extensions) installed on the machine.
- *langNames* receives the names of the languages expressed in the current application language. For example, for a French dictionary, the value "français (France)" is returned on a machine configured in French and "French (France)" on an English system. The language name is followed by "- Hunspell". This field is only valid for files "known" by 4D. For unknown files (for example, custom files), the name "N/A - Hunspell" is returned. This does not prevent you from using the dictionary (if the file concerned is valid), the ID returned may be passed to the **SPELL SET CURRENT DICTIONARY** command.

## Example

You put "fr-classic+reform1990.aff" and "fr-classic+reform1990.dic" as well as "fr-dentist.aff" and "fr-dentist.dic" into the Hunspell directory:

```
ARRAY LONGINT($langID;0)
ARRAY TEXT($dicName;0)
ARRAY TEXT($langDesc;0)
SPELL GET DICTIONARY LIST($langID;$dictName;$langDesc)
```

| $langID | $dictName | $langDesc |
|---|---|---|
| 65536 | en_GB | English (UK) |
| 65792 | en_US | English (USA) |
| 131072 | de_DE | German (Germany) |
| 196608 | es_ES | Spanish |
| 262144 | fr_FR | French (France) |
| 589824 | nb_NO | Norwegian Bokmal (Norway) |
| 1074036166 | fr-classic+reform1990 | French (France) - Hunspell |
| 1073901273 | fr-dentist | No description - Hunspell |

# SPELL SET CURRENT DICTIONARY

SPELL SET CURRENT DICTIONARY ( dictionary )

| Parameter | Type | | Description |
|---|---|---|---|
| dictionary | Longint, Text | ⇒ | ID, Name, or Language code of dictionary to use for spell-check |

## Description

The **SPELL SET CURRENT DICTIONARY** command causes the replacement of the current dictionary with the one specified by the *dictionary* parameter. The current dictionary is used for the built-in spell-check feature in 4D (for more information, refer to the 4D *Design Reference* manual) as well as the one in the 4D Write and 4D View plug-ins. The modification of the current dictionary is reflected immediately in all the processes of the database for the session, as well as in the 4D Write and 4D View plug-in areas.

By default, 4D uses:

- under Windows, the Hunspell dictionary corresponding to the application language,
- under OS X, the native spell checker.

**Note:** For more information about Hunspell dictionaries, refer to the **Support of Hunspell dictionaries** section.

You can use the *dictionary* parameter to change the dictionary. You can pass either:

- a Hunspell dictionary ID number (returned by the **SPELL GET DICTIONARY LIST** command),
- a Hunspell dictionary name (corresponding to the file name of the Hunspell dictionary, with or without its extension),
- a BCP 47, ISO 639-1 or ISO 639-2 language code. For example, with the BCP 47 language code "en-US" indicates American English and "en-GB" specifies British English. These codes are redirected internally to the corresponding current dictionary (Hunspell or native OS X).

**Compatibility note:** In previous versions of 4D, "Cordial" dictionaries were supported. For compatibility, it is still possible to pass a "Cordial" dictionary number in the *dictionary* parameter (value or constant from the "**Dictionaries**" theme). In this case, however, the dictionary is redirected internally to an equivalent Hunspell dictionary (or the native dictionary under OS X).

## System variables and sets

If the *dictionary* is loaded correctly, the system variable OK is set to 1; otherwise, it is set to 0 and an error is returned.

## Example

Loading of the "fr-classic" dictionary found in the Hunspell folder:

```
SPELL SET CURRENT DICTIONARY("fr-classic")
 // SPELL SET CURRENT DICTIONARY ("FR-classic.dic") is valid
```

# SQL

- Overview of SQL Commands
- On SQL Authentication Database Method
- Begin SQL
- End SQL
- Get current data source
- GET DATA SOURCE LIST
- Is field value Null
- QUERY BY SQL
- SET FIELD VALUE NULL
- SQL CANCEL LOAD
- SQL End selection
- SQL EXECUTE
- SQL EXECUTE SCRIPT
- SQL EXPORT DATABASE
- SQL EXPORT SELECTION
- SQL GET LAST ERROR
- SQL GET OPTION
- SQL LOAD RECORD
- SQL LOGIN
- SQL LOGOUT
- SQL SET OPTION
- SQL SET PARAMETER
- START SQL SERVER
- STOP SQL SERVER
- *_o_USE EXTERNAL DATABASE*
- *_o_USE INTERNAL DATABASE*

# Overview of SQL Commands

4D includes an integrated SQL kernel. The program also includes an SQL server that other 4D applications or third-party applications can query (via the 4D ODBC driver).

The SQL documentation in 4D is built upon two main parts:

- The **4D SQL Reference Guide** (**4D SQL Reference**). This manual describes the different ways of accessing the 4D SQL kernel, the configuration of the SQL server as well as the commands and keywords that can be used in SQL queries (for example **SELECT** or **UPDATE**). Please refer to this manual for any question regarding SQL language implementation in 4D.
- The **SQL theme of the "Language" manual** (**SQL**). This theme groups together various 4D high-level commands concerning the use of SQL in 4D:
  - Control of the SQL server: **START SQL SERVER** and **STOP SQL SERVER**
  - Direct access to the integrated SQL kernel: **SET FIELD VALUE NULL**, **Is field value Null**, **QUERY BY SQL**
  - Management of connections to external or internal data sources (SQL pass-through): **GET DATA SOURCE LIST**, **Get current data source**, **SQL LOGIN**, **SQL LOGOUT**.
  - High-level commands for handling data in the framework of direct SQL connections or via ODBC: **Begin SQL**, **End SQL**, **SQL CANCEL LOAD**, **SQL LOAD RECORD**, **SQL EXECUTE**, **SQL End selection**, **SQL SET OPTION**, **SQL SET PARAMETER**, **SQL GET LAST ERROR**, **SQL GET OPTION**.

## How high-level SQL commands work

The built-in SQL commands of 4D begin with the prefix "SQL" and implement the following principles:

- Unless indicated otherwise, you can use these commands with the 4D internal SQL kernel or in an external connection that is opened directly or via ODBC. The **SQL LOGIN** command lets you specify the type of connection to open.
- The scope of a connection is the process. If you want to manage several simultaneous connections, you must start a process by **SQL LOGIN**.
- You can intercept any ODBC errors generated during the execution of one of the high-level SQL commands using the **ON ERR CALL** command. The **SQL GET LAST ERROR** command can be used in this case to obtain additional information.

## Support of standard ODBC

The ODBC (Open DataBase Connectivity) standard specifies a library of standardized functions. These functions allow an application such as 4D to access any ODBC-compatible data management system (databases, spreadsheets, another 4D application, etc.) via SQL language.

**Note:** 4D also allows data to be imported from and exported to an ODBC source via the **IMPORT ODBC** and **EXPORT ODBC** commands or "manually" in Design mode. For more information, please refer to the 4D *Design Reference* manual.

Note: The high-level SQL commands of 4D can be used to implement simple solutions allowing 4D applications to communicate with ODBC data sources. If your applications require more extensive support of ODBC standards, you will need to have the “low level” ODBC plug-in for 4D, **4D ODBC Pro**.

## Correspondence of data types

The following table lists the correspondences that are automatically established by 4D between 4D and SQL data types:

| 4D Type | SQL Type |
|---|---|
| C_STRING | SQL_C_CHAR |
| C_TEXT | SQL_C_CHAR |
| C_REAL | SQL_C_DOUBLE |
| C_DATE | SQL_C_TYPE_DATE |
| C_TIME | SQL_C_TYPE_TIME |
| C_BOOLEAN | SQL_C_BIT |
| C_INTEGER | SQL_C_SHORT |
| C_LONGINT | SQL_C_SLONG |
| C_BLOB | SQL_C_BINARY |
| C_PICTURE | SQL_C_BINARY |
| C_GRAPH | SQL_C_BINARY |

**Note:** Object type data (**C_OBJECT**) is not supported by 4D's SQL kernel.

## Referencing 4D expressions in SQL requests

4D provides two ways for inserting 4D expressions (variables, arrays, fields, pointers, valid expressions) into SQL requests: direct association and the *setting* of parameters using **SQL SET PARAMETER**.

Direct association can be carried out in two ways:

- Insertion of the name of the 4D object between the **<<** and **>>** characters in the text of the request.
- Precede the reference with a colon ":".

```
SQL EXECUTE("INSERT INTO emp (empnum,ename) VALUES (<<vEmpnum>>,<<vEname>>)")
SQL EXECUTE("SELECT age FROM People WHERE name= :vName")
```

**Note:** In compiled mode, you cannot use references to local variables (beginning with the $ symbol).

In these examples, the current values of the 4D vEmpnum, vEname and vName variables will replace the parameters when the request is executed. This solution also works with 4D fields and arrays.

This easy-to-use syntax nevertheless has the drawback of not being compliant with the SQL standard and of not allowing the use of output parameters. To remedy this, you can use the **SQL SET PARAMETER** command. This command can be used to set each 4D object to be integrated into a request as well as its mode of use (input, output or both). The syntax produced is thus standard. For more information, please refer to the description of the **SQL SET PARAMETER** command.

1. This example executes an SQL query that directly uses the associated 4D arrays:

```
ARRAY TEXT(MyTextArray;10)
ARRAY LONGINT(MyLongintArray;10)

For(vCounter;1;Size of array(MyTextArray))
   MyTextArray{vCounter}:="Text"+String(vCounter)
   MyLongintArray{vCounter}:=vCounter
End for
SQL LOGIN("mysql";"root";"")
SQLStmt:="insert into app_testTable (alpha_field, longint_field) VALUES (<<MyTextArray>>, <<MyLongintArray>>)"
SQL EXECUTE(SQLStmt)
```

2. This example can be used to execute an SQL query that directly uses the associated 4D fields:

```
ALL RECORDS([Table 2])
SQL LOGIN("mysql";"root";"")
SQLStmt:="insert into app_testTable (alpha_field, longint_field) VALUES (<<[Table 2]Field1>"+">,<<[Table 2]Field2>>)"
SQL EXECUTE(SQLStmt)
```

3. This example lets you execute an SQL query by directly passing a variable via a dereferenced pointer:

```
C_LONGINT($vLong)
C_POINTER($vPointer)
$vLong:=1
$vPointer:=->$vLong
SQL LOGIN("mysql";"root";"")
```

```
SQLStmt:="SELECT Col1 FROM TEST WHERE Col1=:$vPointer"
SQL EXECUTE(SQLStmt)
```

### Use of local variables in compiled mode

In compiled mode, you can use local variable references (beginning with the $ character) in SQL statements under certain conditions:

- You can use local variables within a **Begin SQL** / **End SQL** sequence, except with the **EXECUTE IMMEDIATE** command;
- You can use local variables with the **SQL EXECUTE** command when these variables are used directly in the parameter of the SQL request and not through references.
  For example, the following code works in compiled mode:

  ```
  SQL EXECUTE("select * from t1 into :$myvar") // works in compiled mode
  ```

  The following code generates an error in compiled mode:

  ```
  C_TEXT(tRequest)
  tRequest:="select * from t1 into :$myvar"
  SQL EXECUTE(tRequest) // error in compiled mode
  ```

## Retrieving values in 4D

Retrieving values in the 4D language that result from SQL queries is carried out in two ways:

- Using the additional parameters of the **SQL EXECUTE** command (recommended solution).
- Using the INTO clause in the SQL query itself (solution reserved for special cases).

## Displaying the result of an SQL query in a list box

It is possible to place the results of an SQL query directly in an array type list box. This offers a rapid means for viewing the results of SQL queries. Only queries of the **SELECT** type can be used. This mechanism cannot be used with an external SQL database.

It works according to the following principles:

- Create the list box which will receive the query results. The data source of the list box must be **Arrays**.
- Execute an SQL query of the **SELECT** type and assign the result to the variable associated with the list box. You can use the **Begin SQL**/**End SQL** keywords (see the 4D *Language Reference* manual).
- List box columns can be sorted or modified by the user.
- Each new execution of a **SELECT** query with the list box leads to the resetting of the columns (it is not possible to fill the same list box progressively using several **SELECT** queries).
- It is recommended to give the list box the same number of columns as there will be in the SQL query result. If the number of list box columns is less than that required by the **SELECT** query, columns are added automatically. If the number of columns is more than required by the **SELECT** query, the unnecessary columns are automatically hidden.
  **Note:** The columns added automatically are bound to **Dynamic variables** of the array type. These dynamic arrays last as long as the form does. A dynamic variable is also created for each header. When the **LISTBOX GET ARRAYS** command is called, the *arrColVars* parameter contains pointers to the dynamic arrays and the *arrHeaderVars* parameter contains pointers to the dynamic header variables. If the added column is, for example, the fifth column, its name is *sql_column5* and its header name is *sql_header5*.
- In interpreted mode, existing arrays that are used by the list box can be retyped automatically according to the data sent by the SQL query.

**Example**

We want to retrieve all the fields of the PEOPLE table and put their contents into the list box having the variable name *vlistbox*. In the object method of a button (for example), simply write:

```
Begin SQL
   SELECT * FROM PEOPLE INTO <<vlistbox>>
End SQL
```

# On SQL Authentication Database Method

---

The **On SQL Authentication Database Method** can be used to filter requests sent to the integrated SQL server of 4D. This filtering can be based on the name and password as well as the (optional) IP address of the user. The developer can use their own table of users or that of the 4D users to evaluate the connection identifiers. Once the connection is authenticated, the **CHANGE CURRENT USER** command must be called in order to control access of requests within the 4D database.

When it exists, the **On SQL Authentication Database Method** is called automatically by 4D or 4D Server on each external connection to the SQL server. The internal system for managing 4D users is therefore not activated. The connection is only accepted if the database method returns **True** in $0 and if the **CHANGE CURRENT USER** command has been executed successfully. If one of these conditions is not met, the request is refused.

**Note**: The statement **SQL LOGIN**(**SQL_INTERNAL**;$user;$password) does not call the **On SQL Authentication Database Method** since it is an internal connection in this case.

The database method receives up to three parameters of the Text type, passed by 4D ($1, $2 and $3), and returns a Boolean, $0. Here is a description of these parameters:

| Parameters | Type | Description |
|---|---|---|
| $1 | Text | User name |
| $2 | Text | Password |
| $3 | Text | (optional) IP address of client at origin of the request(*) |
| $0 | Boolean | True = request accepted, False = request refused |

(*) 4D returns IPv4 addresses in a hybrid IPv6/IPv4 format written with a 96-bit prefix, for example ::ffff:192.168.2.34 for the IPv4 address 192.168.2.34. For more information, refer to the **Support of IPv6** section.

You must declare these parameters as follows:

```
  ` On Web Authentication database method

C_TEXT($1;$2;$3)
C_BOOLEAN($0)
  ` Code for method
```

The password ($2) is received as standard text.

You must check the identifiers of the SQL connection in the **On SQL Authentication Database Method**. For example, you can check the name and password using a custom table of users. If the identifiers are valid, pass **True** in $0 to accept the connection. Otherwise, pass **False** in $0; in this case, the connection is refused.

**Note:** If the **On SQL Authentication Database Method** does not exist, the connection is evaluated using the integrated user management system of 4D (if it is activated, in other words, if a password has been assigned to the Designer). If this system is not activated, users are connected with Designer access rights (free access).

If you have passed **True** in $0, you must then successfully call the **CHANGE CURRENT USER** command in the **On SQL Authentication Database Method** in order for the request to be accepted and for 4D to open an SQL session for the user.

The use of the **CHANGE CURRENT USER** command can be used to implement a virtual authentication system which has the double advantage of allowing the control of connection actions and of hiding the connection identifiers from the outside in the 4D SQL session.

This example of the **On SQL Authentication Database Method** checks whether the connection request comes from the internal network, validates the identifiers and then assigns access rights to the "sql_user" user for the SQL session.

```
C_TEXT($1;$2;$3)
C_BOOLEAN($0)
  `$1: user
  `$2: password
  `{$3: IP address of client}
ON ERR CALL("SQL_error")
If(checkInternalIP($3))
  `The checkInternalIP method checks whether the IP address is internal
   If($1="victor") & ($2="hugo")
      CHANGE CURRENT USER("sql_user";"")
```

```
      If(OK=1)
         $0:=True
      Else
         $0:=False
      End if
   Else
      $0:=False
   End if
Else
   $0:=False
End if
```

# Begin SQL

Begin SQL
Does not require any parameters

## Description

**Begin SQL** is a keyword used in the Method editor to indicate the beginning of a sequence of SQL commands that must be interpreted by the current data source of the process (the integrated SQL engine of 4D or any source specified via the **SQL LOGIN** command).

A sequence of SQL commands started with **Begin SQL** must be closed with the **Begin SQL** keyword.

These keywords work as follows:

- You can place one or more blocks of **Begin SQL/End SQL** tags in the same method. You can generate methods made up entirely of SQL code or mix 4D code and SQL code in the same method.
- You can write several SQL statements on the same line or on different lines by separating them with a semi-colon ";". For example, you can write:

```
Begin SQL
   INSERT INTO SALESREPS (NAME, AGE) VALUES (Henry,40);
   INSERT INTO SALESREPS (NAME, AGE) VALUES (Bill,35)
End SQL
```

or:

```
Begin SQL
   INSERT INTO SALESREPS (NAME, AGE) VALUES (Henry,40);INSERT INTO SALESREPS (NAME, AGE) VALUES (Bill,35)
End SQL
```

Note that the 4D **Debugger** will evaluate the SQL code line by line. In certain cases, it may be preferable to use more than one line.

# End SQL

End SQL

Does not require any parameters

## Description

**End SQL** is a keyword indicating the end of a sequence of SQL commands in the Method editor.

A sequence of SQL statements must be enclosed with the **Begin SQL** and **End SQL** keywords. For more information, refer to the description of the **Begin SQL** keyword.

# Get current data source

Get current data source -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String | ➲ | Name of current data source being used |

## Description

The **Get current data source** command returns the name of the current data source of the application. The current data source receives the SQL queries executed within **Begin SQL/End SQL** structures.

When the current data source is the local 4D database, the command returns the string “;DB4D_SQL_LOCAL;”, which corresponds to the value of the SQL_INTERNAL constant ("**SQL**" theme).

This command lets you check the current data source, generally before executing an SQL query.

# GET DATA SOURCE LIST

GET DATA SOURCE LIST ( sourceType ; sourceNamesArr ; driversArr )

| Parameter | Type | | Description |
|---|---|---|---|
| sourceType | Longint | ➡ | Source type: user or system |
| sourceNamesArr | Text array | ⬅ | Array of data source names |
| driversArr | Text array | ⬅ | Array of drivers for sources |

## Description

The **GET DATA SOURCE LIST** command returns, in the *sourceNamesArr* and *driversArr* arrays, the names and drivers of the *sourceType* type data sources defined in the ODBC manager of the operating system.

4D allows you to connect to an external ODBC data source directly via the language and execute SQL queries within a **Begin SQL/End SQL** tag structure. This works as follows: the **GET DATA SOURCE LIST** command can be used to get a list of data sources present on the machine. The **SQL LOGIN** command can then be used to designate the source to be used. You can then execute SQL queries using a **Begin SQL/End SQL** tag structure in the "current" source. To carry out queries using the 4D internal engine again, simply pass the **SQL LOGOUT** command. For more information about SQL commands in the Method editor, please refer to the **4D SQL Reference** manual.

In *sourceType*, pass the type of data source that you want to retrieve. You can use one of the following constants, found in the "**SQL**" theme:

| Constant | Type | Value |
|---|---|---|
| System data source | Longint | 2 |
| User data source | Longint | 1 |

**Note:** This command does not take file type data sources into account.

The command fills and sizes the *sourceNamesArr* and *driversArr* arrays with the corresponding values.

**Note:** If you want to connect to an external 4D data source via ODBC, you will need to have installed the 4D ODBC Driver on your machine. For more information, please refer to the 4D ODBC Driver Installation manual.

## Example

Example using a user data source:

```
ARRAY TEXT(arrDSN;0)
ARRAY TEXT(arrDSNDrivers;0)
GET DATA SOURCE LIST(User data source;arrDSN;arrDSNDrivers)
```

## System variables and sets

If the command is executed correctly, the OK system variable is set to 1. Otherwise, it is set to 0 and an error is generated.

# Is field value Null

Is field value Null ( aField ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ⇒ | Field to be evaluated |
| Function result | Boolean | ⇐ | True = field is NULL, False = field is not NULL |

## Description

The **Is field value Null** command returns **True** if the field designated by the *aField* parameter contains the NULL value, and **False** otherwise.

The NULL value is used by the SQL kernel of 4D. For more information, refer to the **4D SQL Reference** manual.

The value returned by this command is only meaningful if the "" option is not checked in the field definition of the Structure editor. Otherwise, it always returns **False**.

# QUERY BY SQL

QUERY BY SQL ( {aTable ;} sqlFormula )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table in which to return a selection of records or Default table if this parameter is omitted |
| sqlFormula | String | ⇒ | Valid SQL search formula representing the WHERE clause of the SELECT query |

## Description

The **QUERY BY SQL** command can be used to take advantage of the SQL kernel integrated into 4D. It can execute a simple SELECT query that can be written as follows:

```
SELECT *
   FROM table
   WHERE <sqlFormula>
```

*aTable* is the name of the table passed in the first parameter and *sqlFormula* is the query string passed in the second parameter.

For example, the following statement:

```
([Employees];"name=' smith' ")
```

is equivalent to the following SQL query:

```
SELECT*FROM Employees WHERE"name=' smith' "
```

The **QUERY BY SQL** command is similar to the **QUERY BY FORMULA** command. It looks for records in the specified table. It changes the current selection of *aTable* for the current process and makes the first record of the new selection the current record.

**Note:** The **QUERY BY SQL** command cannot be used in the context of an external SQL connection; it connects directly to the integrated SQL engine of 4D.

**QUERY BY SQL** applies *sqlFormula* to each record in the table selection. *sqlFormula* is a Boolean expression that must return **True** or **False**. As you may know, in the SQL standard, a search condition can yield a **True**, **False** or NULL result. All the records (rows) where the search condition returns **True** are included in the new current selection.

The *sqlFormula* expression may be simple, such as comparing a field (column) to a value; or it may be complex, such as performing a calculation. Like **QUERY BY FORMULA**, **QUERY BY SQL** is able to evaluate information in related tables (see example 4). *sqlFormula* must be a valid SQL statement that is compliant with the SQL-2 standard and with respect to the limitations of the current SQL implementation of 4D. For more information about SQL support in 4D, refer to the **4D SQL Reference** manual.

The *sqlFormula* parameter can use references to 4D expressions. The syntax to use is the same as for the integrated SQL commands or the code included between the **Begin SQL**/**End SQL** tags, i.e.: *<<MyVar>>* or *:MyVar*.

**Note:** This command is compatible with the **SET QUERY LIMIT** and **SET QUERY DESTINATION** commands.

**Reminder:** You cannot have references to local variables in compiled mode. For more information about SQL programming in 4D, refer to the section **Overview of SQL Commands**.

### About Relations

**QUERY BY SQL** does not use relations between tables defined in the 4D Structure editor. If you want to make use of related data, you will have to add a JOIN to the query. For example, assuming we have the following structure with a Many-to-One relation from[Persons]City to [Cities]Name:

```
[People]
   Name
   City
[Cities]
   Name
   Population
```

Using the **QUERY BY FORMULA** command, you can write:

```
QUERY BY FORMULA([People];[Cities]Population>1000)
```

Using **QUERY BY SQL**, you must write the following statement, regardless of whether the relation exists:

```
QUERY BY SQL([People];"people.city=cities.name AND cities.population>1000")
```

**Note: QUERY BY SQL** handles One-to-Many and Many-to-Many relations differently than **QUERY BY FORMULA**.

## Example 1

This example shows the offices where sales exceed 100. The SQL query is:

```
SELECT *
   FROM Offices
   WHERE Sales > 100
```

When using the **QUERY BY SQL** command:

```
C_STRING(30;$queryFormula)
$queryFormula:="Sales > 100"
QUERY BY SQL([Offices];$queryFormula)
```

## Example 2

This example shows the orders that fall into the 3000 to 4000 range. The SQL query is:

```
SELECT *
   FROM Orders
   WHERE Amount BETWEEN 3000 AND 4000
```

When using the **QUERY BY SQL** command:

```
C_STRING(40;$queryFormula)
$queryFormula:="Amount BETWEEN 3000 AND 4000"
QUERY BY SQL([Orders];$queryFormula)
```

## Example 3

This example shows how to get the query result ordered by a specific criterion. The SQL query is:

```
SELECT *
   FROM People
   WHERE City =' Paris'
      ORDER BY Name
```

When using the **QUERY BY SQL** command:

```
C_STRING(40;$queryFormula)
$queryFormula:="City=  'Paris'  ORDER BY Name"
QUERY BY SQL([People];$queryFormula)
```

## Example 4

This example shows a query using related tables in 4D. In SQL you should use a JOIN to simulate this relation. Assuming we have the two following tables:

```
[Invoices] with the following columns (fields):
   ID_Inv: Longint
   Date_Inv: Date
   Amount: Real
```

```
[Lines_Invoices] with the following columns (fields):
   ID_Line: Longint
   ID_Inv: Longint
   Code: Alpha (10)
```

There is a Many-to-One relation from [Lines_Invoices]ID_Inv to [Invoices]ID_Inv.
Using the **QUERY BY FORMULA** command, you could write:

```
QUERY BY FORMULA([Lines_Invoices];([Lines_Invoices]Code="FX-200") & (Month of([Invoices]Date_Inv)=4))
```

The SQL query is:

```
SELECT ID_Line
   FROM Lines_Invoices, Invoices
   WHERE Lines_Invoices.ID_Inv=Invoices.ID_Inv
      AND Lines_Invoices.Code='FX-200'
      AND MONTH(Invoices.Date_Inv) = 4
```

When using the **QUERY BY SQL** command:

```
C_STRING(40;$queryFormula)
$queryFormula:="Lines_Invoices.ID_Inv=Invoices.ID_InvAND Lines_Invoices.Code=' FX-200'  AND MONTH(Invoices.Date_Inv)=4"
QUERY BY SQL([Lines_Invoices];$queryFormula)
```

## System variables and sets

If the format of the search condition is correct, the system variable OK is set to 1. Otherwise, it is set to 0, the result of the command is an empty selection and an error is returned. This error can be intercepted by a method installed using the **ON ERR CALL** command.

# SET FIELD VALUE NULL

SET FIELD VALUE NULL ( aField )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ⇒ | Field where NULL value is to be attributed |

## Description

The **SET FIELD VALUE NULL** command assigns the NULL value to the field designated by the *aField* parameter.

The NULL value is used by the SQL kernel of 4D. For more information, please refer to the 4D SQL Reference manual.

**Notes:**

- It is possible to disallow the Null value for 4D fields at the Structure editor level (see the Design Reference manual).
- **SET FIELD VALUE NULL** erases the contents of object fields.

# SQL CANCEL LOAD

SQL CANCEL LOAD
Does not require any parameters

## Description

The **SQL CANCEL LOAD** command ends the current SELECT request and initializes the parameters.
This command is used to execute several SELECT requests within the same connection (i.e. the same cursor) initiated by the **SQL LOGIN** command.

## Example

In this example, two requests are executed in the same connection:

```
C_BLOB(Myblob)
C_TEXT(MyText)
SQL LOGIN("mysql";"root";"")

SQLStmt:="SELECT blob_field FROM app_testTable"
SQL EXECUTE(SQLStmt;Myblob)
While(Not(SQL End selection))
   SQL LOAD RECORD
End while

 `Resetting of cursor
SQL CANCEL LOAD

SQLStmt:="SELECT Name FROM Employee"
SQL EXECUTE(SQLStmt;MyText)
While(Not(SQL End selection))
   SQL LOAD RECORD
End while
```

## System variables and sets

If the command has been correctly executed, the system variable OK returns 1. Otherwise, it returns 0.

# SQL End selection

SQL End selection -> Function result

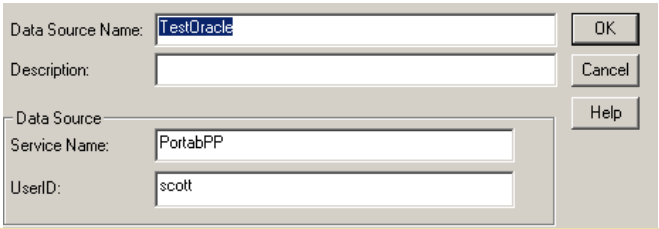
| Parameter | Type | Description |
|---|---|---|
| Function result | Boolean | Result set boundaries reached |

## Description

The **SQL End selection** command is used to determine if the boundaries of the result set have been reached.

## Example

The code below connects to an external data source (Oracle) using the following parameters:




```
C_TEXT(vName)

SQL LOGIN("TestOracle";"scott";"tiger")
If(OK=1)
   SQL EXECUTE("SELECT ename FROM emp";vName)
   While(Not(SQL End selection))
      SQL LOAD RECORD
   End while
   SQL LOGOUT
End if
```

This code will return in the 4D *vName* variable the emp names (ename) stored in the table named emp.

# SQL EXECUTE

SQL EXECUTE ( sqlStatement {; boundObj}{; boundObj2 ; ... ; boundObjN} )

| Parameter | Type | | Description |
|---|---|---|---|
| sqlStatement | Text | ⇒ | SQL command to execute |
| boundObj | Variable, Field | ⇐ | Receives result (if necessary) |

## Description

The **SQL EXECUTE** command executes an SQL command and binds the result to 4D objects (arrays, variables or fields).

A valid connection must be specified in the current process in order to execute this command.

The *sqlStatement* parameter contains the SQL command to execute. *boundObj* receives the results. Variables are bound in the column sequence order, which means that any remaining remote columns are discarded.

If 4D fields are passed as parameters in *boundObj*, the command will create records and save them automatically. 4D fields must come from the same table (a field from table 1 and a field from table 2 cannot be passed in the same call). If fields from more than one table are passed, an error is generated.

**Warning:** When you pass 4D fields in the *boundObj* parameter(s) and execute the **SELECT** command, it is always the data of the remote 4D source that is modified. If you want to retrieve data from the remote source locally, you must use intermediary local arrays and call the **INSERT** command (see example 6).

If you pass 4D arrays in the *boundObj* parameter(s), it is advisable to declare them before calling the command in order to check the type of data processed. Arrays are automatically resized when necessary.

With a 4D variable, one record is fetched at a time. The other results are ignored.

**Note:** For more information about referencing 4D expressions in SQL queries, refer to the **Overview of SQL Commands** section.

## Example 1

In this example, we will get the ename column of the emp table of the data source. The result is stored in the [Employee]Name 4D field. 4D records will be created automatically:

```
SQLStmt:="SELECT ename FROM emp"
SQL EXECUTE(SQLStmt;[Employee]Name)
SQL LOAD RECORD(SQL all records)
```

## Example 2

To check the creation of records, it is possible to include code within a transaction and to validate it only if the operation proves to be satisfactory:

```
SQL LOGIN("mysql";"root";"")
SQLStmt:="SELECT alpha_field FROM app_testTable"
START TRANSACTION
SQL EXECUTE(SQLStmt;[Table 2]Field1)
While(Not(SQL End selection))
   SQL LOAD RECORD
   ... //Place the data validation code here
End while
VALIDATE TRANSACTION //Validation of the transaction
```

## Example 3

In this example, we want to get the ename column of the emp table of the data source. The result will be stored in an *aName* array. We fetch records 10 at a time.

```
ARRAY STRING(30;aName;20)
SQLStmt:="SELECT ename FROM emp"
SQL EXECUTE(SQLStmt;aName)
While(Not(SQL End selection))
   SQL LOAD RECORD(10)
End while
```

## Example 4

In this example, we want to get the ename and job of the emp table for a specific ID (WHERE clause) of the data source. The result will be stored in the *vName* and *vJob* 4D variables. Only the first record is fetched.

```
SQLStmt:="SELECT ename, job FROM emp WHERE id = 3"
SQL EXECUTE(SQLStmt;vName;vJob)
SQL LOAD RECORD
```

## Example 5

In this example, we want to get the Blob_Field column of the Test table in the data source. The result will be stored in a BLOB variable whose value is updated each time a record is loaded.

```
C_BLOB(MyBlob)
SQL LOGIN
SQL EXECUTE("SELECT Blob_Field FROM Test";MyBlob)
While(Not(SQL End selection))
 //We look through the results
   SQL LOAD RECORD
 //The value of MyBlob is updated on each call
End while
```

## Example 6

You want to retrieve data locally from a remote 4D Server database where it is stored. To do this, you must use intermediary arrays:

```
 // Log in to the remote database
SQL LOGIN("IP:192.168.18.15:19812";"user";"password";*)
If(OK=1)
 // Starting from here all SQL requests are made on the remote database
   C_TEXT($LastName_value) // 4D variable used in the search statement
   ARRAY TEXT($a_LastName;0) // Temporary storage of remote values for LastName
   ARRAY TEXT($a_FirstName;0) // Temporary storage of remote values for FirstName
   C_BOOLEAN($UseSQL) //Choice of means for local storage of data from the remote database
 // (demo only)

   $LastName_value:="Smith" // Initialization of 4D variable

 // Associate the 4D $LastName_value variable with the first "?" in the SQL request
   SQL SET PARAMETER($LastName_value;SQL param in)

 // From the remote PERSONS table, retrieve the values of the LastName and FirstName fields
 // where "LastName = Smith" and store them in the $a_LastName and $a_FirstName arrays
   SQL EXECUTE("SELECT LastName, FirstName FROM PERSONS WHERE LastName = ?";$a_LastName;$a_FirstName)
   If(Not(SQL End selection)) // If at least one record is found

      SQL LOAD RECORD(SQL all records) // Load all the records

      $UseSQL:=True // Chooses the way to integrate the data (demo only)

      If($UseSQL) // Use SQL requests
         SQL LOGOUT // Log out from the remote database
         SQL LOGIN(SQL_INTERNAL;"user";"password") // Log in to the local database
```

```
    // Starting from here all SQL requests are made on the local database
    // Save the $a_LastName and $a_FirstName arrays in the local PERSONS table
            SQL EXECUTE("INSERT INTO PERSONS(LastName, FirstName) VALUES (:$a_LastName, :$a_FirstName);")

         Else // Using 4D commands
            For($i;1;Size of array($a_LastName))
               CREATE RECORD([PERSONS])
               [PERSONS]LastName:=$a_LastName{$i}
               [PERSONS]FirstName:=$a_FirstName{$i}
               SAVE RECORD([PERSONS])
            End for
         End if
      End if
      SQL LOGOUT // Close the connection
   End if
```

## System variables and sets

If the command has been executed correctly, the system variable OK returns 1. Otherwise, it returns 0.

# SQL EXECUTE SCRIPT

SQL EXECUTE SCRIPT ( scriptPath ; errorAction {; attribName ; attribValue} {; attribName2 ; attribValue2 ; ... ; attribNameN ; attribValueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| scriptPath | Text | ⇒ | Complete pathname of file containing SQL script to execute |
| errorAction | Longint | ⇒ | Action to carry out in case of error during script execution |
| attribName | Text | ⇒ | Name of attribute to use |
| attribValue | Text | ⇒ | Value of attribute |

## Description

The **SQL EXECUTE SCRIPT** command is used to execute a series of SQL statements placed in the script file designated by *scriptPath*. This command can only be executed on a local machine (local 4D or stored procedure on 4D Server). It works with the current database (internal or external database).

**Note:** This command cannot be used with an external connection that is opened directly or via ODBC.

Pass the complete pathname of the text file containing the SQL statements to be executed in the *scriptPath* parameter. The pathname must be expressed using the syntax of the current system. If you pass an empty string ("") in *scriptPath*, a standard Open document dialog box will be displayed so that the user can select the script file to be executed.

**Note:** The **SQL EXPORT DATABASE** and **SQL EXPORT SELECTION** commands automatically generate this script file.

The *errorAction* parameter is used to configure the functioning of the command when an error occurs during script execution. You can pass one of the three following constants, placed in the **SQL** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| SQL On error abort | Longint | 1 | In the event of an error, 4D immediately stops script execution. |
| SQL On error confirm | Longint | 2 | In the event of an error, 4D displays a dialog box describing the error and allowing the user to interrupt or continue script execution. |
| SQL On error continue | Longint | 3 | In the event of an error, 4D ignores it and continues script execution. |

The *attribName* and *attribValue* parameters must be passed by pairs. These parameters are intended to be used to specify specific attributes for the script execution. In the current version of 4D, a single attribute can be passed in *attribName*, available via the following constant, placed in the **SQL** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| SQL use access rights | String | SQL_Use_Access_Rights | Used to restrict the access rights to be applied during execution of the SQL commands of the script. When you use this attribute, you must pass 0 or 1 in *attribValue*:<br>• *attribValue* = 1: 4D uses the access rights of the current 4D user.<br>• *attribValue* = 0 (or attribute not specified): 4D does not restrict access, the Designer rights are used. |

If the 4D log file is activated (via the selectors 28 or 45 of the **SET DATABASE PARAMETER** command), each SQL command executed will generate an entry with the following information:

- Type of SQL command
- Number of records affected by the command
- Duration of command execution
- For each error encountered:
  - the error code
  - the error text if available

If the script is executed correctly (no error occurs), the *OK* system variable is set to 1. In the event of an error, the *OK* system variable is set to 0 or not according to the *errorAction* parameter:

- If *errorAction* is SQL On error abort (value 1), *OK* is set to 0.

- If *errorAction* is SQL On error confirm (value 2), the *OK* variable is set to 0 if the user chooses to stop the operation and 1 if they choose to continue .
- If *errorAction* is SQL On error continue (value 3), the *OK* variable is always 1.

**Note:** If you use this command to execute memory-consuming actions such as massive data imports, you can consider calling the **ALTER DATABASE** SQL command in order to temporarily disable the SQL options.

# SQL EXPORT DATABASE

SQL EXPORT DATABASE ( folderPath {; numFiles {; fileLimitSize {; fieldLimitSize}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| folderPath | Text | ⇒ | Pathname of export folder or "" to display folder selection dialog box |
| numFiles | Longint | ⇒ | Maximum number of files per folder |
| fileLimitSize | Longint | ⇒ | Size limit value of export files (in KB) |
| fieldLimitSize | Longint | ⇒ | Size limit (in bytes) below which the contents of a Text, BLOB or Picture field is embedded into the main file |

## Description

The **SQL EXPORT DATABASE** command exports in SQL format all the records of all the tables in the database. In SQL, this global export operation is called "Dump".

**Note:** This command cannot be used with an external connection that has been opened directly or via ODBC.

For each table, the command generates a text file containing the SQL statements necessary for importing data into another database. This file can be used directly by the **SQL EXECUTE SCRIPT** command in order to import data into another 4D database.

The export files will be placed in a folder named "SQLExport" that is created in the destination folder designated by the *folderPath* parameter. Please note that if an "SQLExport" folder already exists at the location specified, the command will replace it without any warning message being displayed.
If you pass an empty string in this parameter, 4D displays a standard dialog box which lets the user designate the destination folder. By default, the dialog box displays the current folder of the user that opened the session ("My Documents" under Windows or "Documents" under Mac OS).

For each exported table, the command carries out the following actions:

- a subfolder with the table name is created in the destination folder.
- a text file named "Export.sql" is created in the subfolder. This file is encoded in UTF-8 with a BOM (byte-order mark). It contains SQL **INSERT** orders corresponding to the exported data. The field values are separated by colons. There may be fewer values than there are fields in the table. In this case, the remaining fields will be considered NULL.
- if the table contains BLOB, picture or text fields (texts stored externally, in other words, outside of records), by default the command creates an additional subfolder named "BLOBS" next to the "Export.sql" file and creates as many subfolders named "BlobsX" as necessary. These subfolders will store as separate files the contents of all the BLOB, picture or external text fields of the table records. The BLOB files are named "BlobXXXXX.BLOB", the text files are named "TEXTXXXXX.TXT"(where XXXXX is a unique number generated by the application). The picture files are named PICTXXXXX.ZZZZ (where XXXXX is a unique number generated by the application and ZZZZ is the extension). When possible, pictures are exported in their original native format with the extension corresponding to the picture type (.jpg, .png, etc.). If the export is not possible in the native format, the pictures are exported in the internal 4D format 4D with the .4PCT extension.
  This default behavior can be adjusted according to the size of the data contained in the field using the optional *fieldLimitSize* parameter (see below).
  **Note:** This behavior differs when you execute **SQL EXPORT DATABASE** from a 4D in remote mode. In this context, the data to be stored externally are included automatically in the "Export.sql" file.

If you pass the *numFiles* parameter, the command will create as many "BlobsX" subfolders as necessary so that each one of them does not contain more than *numFiles* BLOB, picture or external text files. By default, if the *numFiles* parameter is omitted, the command limits the number of files to 200. If you pass 0, each subfolder will contain at least one file.

The *fileLimitSize* parameter lets you set a size limit (in KB) for each "Export.sql" created on disk. Once the size of the export file being created reaches the value set in *fileLimitSize*, 4D stops writing records, closes the file and creates a new file named "ExportX.sql" (where X represents the sequence number) next to the previous one. Note that this is a theoretical limit: the actual size of the "ExportX.sql" files exceeds the value set by *fileLimitSize* because the file is only closed after the record that was being exported when the limit was reached has been completely written (the contents of the records is not divided). The minimum value accepted is 100 and the maximum value (default value) is 100,000 (100 MB).

The optional *fieldLimitSize* parameter sets a size limit below which the contents of an external BLOB, Picture or Text field will be embedded in the main "Export.sql" file rather than saved as a separate file. This purpose of this parameter is to optimize export operations by limiting the number of subfolders and files created on disk.
This parameter must be expressed in bytes. For example, if you pass 1000, any external BLOB, Picture or Text fields that

contain data with a size less than or equal to 1000 bytes are embedded into the main export file.
Note that binary field data (BLOB and Picture) that are embedded into the export file are written in hexadecimal format, in the form of X'0f20' (standard SQL hexadecimal notation, see **literal**). This format is automatically supported by the 4D SQL engine.
By default, if the *fieldLimitSize* parameter is omitted, external BLOB, Picture and Text fields are always exported as external files regardless of their size.

In the export file, there may be fewer values than there are fields in the table. In this case, the empty fields will be considered as NULL. You can also pass the NULL value in a field.

If the export has been carried out correctly, the OK variable is set to 1. Otherwise, it is set to 0.

**Note:** This command does not support Object type fields.

# SQL EXPORT SELECTION

SQL EXPORT SELECTION ( aTable ; folderPath {; numFiles {; fileLimitSize {; fieldLimitSize}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table from which to export selection |
| folderPath | Text | ⇒ | Pathname of export folder or "" to display folder selection dialog box |
| numFiles | Longint | ⇒ | Maximum number of files per folder |
| fileLimitSize | Longint | ⇒ | Maximum size of Export.sql file (in KB) |
| fieldLimitSize | Longint | ⇒ | Size limit (in bytes) below which the contents of a Text, BLOB or Picture field are embedded into the main file |

## Description

The **SQL EXPORT SELECTION** command exports in SQL format the records of the current selection of the 4D table designated by the *aTable* parameter.

This command is nearly identical to the **SQL EXPORT DATABASE** command. The file generated can be used directly by the **SQL EXECUTE SCRIPT** command in order to import data into another 4D database. The main difference between these two commands is that **SQL EXPORT SELECTION** only exports the current selection of *aTable* whereas **SQL EXPORT DATABASE** exports the entire database. Also, unlike the **SQL EXPORT DATABASE** command, this command does not work with external SQL databases. It can only be used with the main database.

Refer to the description of the **SQL EXPORT DATABASE** command for a detailed description of the functioning and parameters of these commands.

If the current selection is empty, the command does nothing. Note that in this case, the destination folder is not emptied.

If the export is carried out correctly, the *OK* variable is set to 1. Otherwise, it is set to 0.

**Note:** This command does not support Object type fields.

# SQL GET LAST ERROR

SQL GET LAST ERROR ( errCode ; errText ; errODBC ; errSQLServer )

| Parameter | Type | | Description |
|---|---|---|---|
| errCode | Longint | ⇦ | Error code |
| errText | Text | ⇦ | Error text |
| errODBC | Text | ⇦ | ODBC error code |
| errSQLServer | Longint | ⇦ | SQL server native error code |

## Description

The **SQL GET LAST ERROR** command returns information related to the last error encountered during the execution of an ODBC command. The error may come from the 4D application, the network, the ODBC source, etc.

This command must generally be called in the context of an error-handling method installed using the **ON ERR CALL** command.

- The *errCode* parameter returns the error code.
- The *errText* parameter returns the error text.

The last two parameters are only filled when the error comes from the ODBC source; otherwise, they are returned empty.

- The *errODBC* parameter returns the ODBC error code (SQL state).
- The *errSQLServer* parameter returns the SQL server native error code.

# SQL GET OPTION

SQL GET OPTION ( option ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ⇒ | Option number |
| value | Longint, Text | ⇐ | Option value |

## Description

The **SQL GET OPTION** command returns the current *value* of the option passed in *option*.

For more information on the different options and their associated values, refer to the description of the **SQL SET OPTION** command.

## System variables and sets

If the command was properly executed, the system variable OK is set to 1. Otherwise, it is set to 0.

# SQL LOAD RECORD

SQL LOAD RECORD {( numRecords )}

| Parameter | Type | | Description |
|---|---|---|---|
| numRecords | Longint | ⇨ | Number of records to load |

## Description

The **SQL LOAD RECORD** command retrieves one or more record(s) in 4D coming from the data source open in the current connection.

The optional *numRecords* parameter sets the number of records to retrieve:

- If you omit this parameter, the command retrieves the current record from the data source. This principle corresponds to the retrieval of data in a loop where one record is received at a time.
- If you pass an integer value in *numRecords*, the command retrieves *numRecords* records.
- If you pass the SQL All Records constant (value -1), the command retrieves all the records of the table.

**Note:** These last two settings are only useful when the retrieved data is associated with 4D arrays or fields.

## System variables and sets

If the command has been executed correctly, the system variable OK returns 1. Otherwise, it returns 0.

# SQL LOGIN

SQL LOGIN {( dataEntry ; userName ; password ; * )}

| Parameter | Type | | Description |
|---|---|---|---|
| dataEntry | String | ⇒ | Publication name of 4D database or<br>IP address of remote database or<br>Name of the data source entry in the ODBC Manager or<br>"" to display the selection dialog box |
| userName | String | ⇒ | Name of the user registered in the data source |
| password | String | ⇒ | Password of the user registered in the data source |
| * | Operator | ⇒ | Applied to Begin SQL/End SQL If omitted: do not apply (local database); if passed: apply |

## Description

The **SQL LOGIN** command allows you to connect to an SQL data source specified in the *dataEntry* parameter. It designates the target of the SQL queries executed subsequently in the current process:

- via the **SQL EXECUTE** command,
- via code placed within the *Begin SQL / End SQL* tags (if the * parameter is passed).

The SQL data source can either be:

- an external 4D Server database that you access directly,
- an external ODBC source,
- the local 4D database (internal database).

In *dataEntry*, you can pass one of the following values: an IP address, a 4D database publication name, an ODBC data source name, an empty string or the SQL_INTERNAL constant.

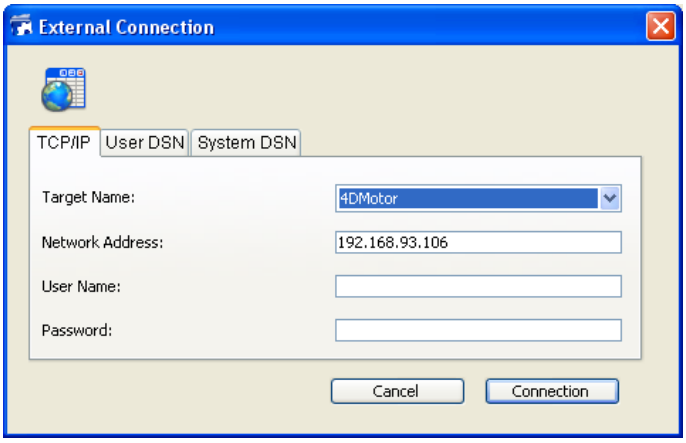
- **IP address**
  Syntax: **IP:<IPAddress>{:<TCPPort>}**
  In this case, the command opens a direct connection with the 4D Server database executed on the machine with the IP address specified. On the "target" machine, the SQL server must be started. If you pass a TCP port number, it must have been specified as the publication port of the SQL server in the "target" database. If you do not pass a TCP port number, the default port will be used (19812). The TCP port number of the SQL server can be modified on the "SQL" page of the Database Settings. Refer to examples 4 and 5.
  If you have enabled SSL for the "target" SQL server (option available in the Database Settings), you must add the ":ssl" keyword to the end of the IP address and TCP port number (mandatory in that case) in order for the server to be able to handle the request correctly (see example 6).

- **4D database publication name**
  Syntax: **4D:<Publication_Name>**
  In this case, the command opens a direct connection with the 4D Server database whose publication name on the network corresponds to the name specified. The network publication name of a database is set on the "Client-Server" page of the Database Settings.
  Refer to example 4.
  **Note:** The TCP port number of the target 4D SQL server (that publishes the 4D database) and the TCP port number of the SQL server of the 4D application that opens the connection must be the same.

- **valid ODBC data source name**
  Syntax: **ODBC:<My_DSN> or <My_DSN>**
  In this case, the *dataEntry* parameter contains the name of the data source as it has been set in the ODBC driver manager.
  **Notes:**
  - For compatibility with previous versions of 4D, it is possible to omit the "ODBC:" prefix. However, for better code readability, it is recommended to use this prefix.
    Refer to example 2.
  - Under Windows, the data source name is case sensitive. For example, if the data source was defined as "4D_v16", passing the value "4D_V16" will fail.

- Under Windows and Mac, the "ODBC:" prefix must be entered using uppercase letters. If you pass "odbc:", the connection will fail.

- **empty string**
  Syntax: ""
  In this case, the command displays the connection dialog box so that the data source to be connected to can be entered manually:




  This dialog box includes several pages. The TCP/IP page includes the following elements:
  - Target Name: This menu is built using two lists:
    - The list of databases that have been opened recently in direct con-nection. The mechanism for updating this list is the same as that of the 4D application, except that the folder containing the .4DLink files is named "Favorites SQL vXX" instead of "Favorites vXX".
    - The list of 4D Server applications whose SQL server is started and whose TCP port for SQL connections is the same as that of the source application. This list is dynamically updated on each new call to the **SQL LOGIN** command without the *dataEntry* parameter. If the "^" character is placed before a database name, this indicates that the connection has been made in secured mode via SSL.
  - Network Address: This area displays the IP address and possibly the TCP port of the database selected in the Target Name menu. You can also enter an IP address in this area and then click on the Connection button in order to connect to the corresponding 4D Server database. You can also specify the TCP port by entering a colon (:) followed by the port number after the address. For example: 192.168.93.105:19855
  - User Name and Password: These areas can be used to enter the con-nection identifiers.
  - The User DSN and System DSN pages display, respectively, the list of user and system ODBC data sources specified in the ODBC driver of the machine. These pages can be used to select a data source and enter the identifiers in order to open a connection with an external ODBC data source.

  If the connection is established, the OK system variable is set to 1. Otherwise, it is set to 0 and an error is generated. This error can be intercepted via an error-handling method installed by the **ON ERR CALL** command.

- **SQL_INTERNAL constant**
  Syntax: SQL_INTERNAL
  In this case, the command redirects subsequent SQL queries to the internal 4D database.

**Warning:** The prefixes used in the *dataEntry* parameter (IP, ODBC, 4D) must be written in uppercase.

*userName* contains the name of the user authorized to connect to the external data source. For example, with Oracle®, the user name can be “Scott”.

*password* contains the password of the user authorized to connect to the external data source. For example, with Oracle®, the password can be “tiger”.

**Note:** In the case of a direct connection, if you pass empty strings in the *userName* and *password* parameters, the connection will only be accepted if 4D passwords are not activated in the target database. Otherwise, the connection will be refused.

The optional * parameter can be used to change the target of the SQL code executed within the *Begin SQL/End SQL* tags. If you do not pass this parameter, the code placed within the *Begin SQL/End SQL* tags will still be sent to the internal SQL engine of 4D, without taking the configuration specified by the **SQL LOGIN** command into account. If you do pass this parameter, the SQL code executed within the *Begin SQL/End SQL* tags will be sent to the source specified in the *dataEntry* parameter.

To close the current connection and free the memory, simply execute the **SQL LOGOUT** command. All the SQL queries are then sent to the internal 4D SQL database.
If you call **SQL LOGIN** again without having explicitly closed the current connection, it will be closed automatically.

**Note:** In the case where an external connection attempt via **SQL LOGIN** fails, the internal 4D database automatically becomes the current data source.

These parameters are optional; if no parameters are passed, the command will bring up the ODBC Login dialog box that allows you to select the external data source.

The scope of this command is per process; in other words, if you want to execute two distinct connections, you must create two processes and execute each connection in each process.

**Warning:** It is not possible to open an ODBC connection in the contexts described below. These configurations lead to blocking of the application:

- connection via ODBC from the running application to itself
- connection via ODBC from a 4D application to 4D Server when a standard client/server connection is already open between these two applications.

## Example 1

This statement will bring up the ODBC Manager dialog box:

```
SQL LOGIN
```

## Example 2

Opening of a connection via the ODBC protocol with the "MyOracle" external data source. SQL queries executed via the **SQL EXECUTE** command and queries included within the **Begin SQL**/**End SQL** tags will be redirected to this connection:

```
SQL LOGIN("ODBC:MyOracle";"Scott";"tiger";*)
```

## Example 3

Open a connection with the 4D internal SQL kernel:

```
SQL LOGIN(SQL_INTERNAL;$user;$password)
```

## Example 4

Opening of a direct connection with the 4D Server application executed on the machine having the IP address 192.168.45.34 and replying on the default TCP port. The SQL queries executed via the **SQL EXECUTE** command will be redirected to this con-nection; the queries included within the **Begin SQL**/**End SQL** tags will not be redirected.

```
SQL LOGIN("IP:192.168.45.34";"John";"azerty")
```

## Example 5

Opening of a direct connection with the 4D Server application executed on the machine having the IP address 192.168.45.34 and replying on TCP port 20150. The SQL queries executed via the **SQL EXECUTE** command and the queries included within the **Begin SQL**/**End SQL** tags will be redirected to this connection.

```
SQL LOGIN("IP:192.168.45.34:20150";"John";"azerty";*)
```

## Example 6

Opening of a direct connection in SSL with the 4D Server application running on the machine with the IP address 192.168.45.34 and responding on the default TCP port. You must have enabled SSL for the SQL server on the 4D Server application:

```
SQL LOGIN("IP:192.168.45.34:19812:ssl";"Admin";"sd156") // Note the ":ssl" after of the IP address and TCP port
```

## Example 7

Opening of a direct connection with the 4D Server application which publishes, on the local network, a database whose publication name is "Accounts_DB." The TCP port used for the SQL server of both databases (set on the SQL page of the Database Settings) must be the same (19812 by default). The SQL queries executed via the **SQL EXECUTE** command will be redirected to this connection; the queries included within the **Begin SQL**/**End SQL** tags will not be redirected.

```
SQL LOGIN("4D:Accounts_DB";"John";"azerty")
```

## Example 8

This example illustrates the connection possibilities provided by the **SQL LOGIN** command:

```
ARRAY TEXT(aNames;0)
ARRAY LONGINT(aAges;0)
SQL LOGIN("ODBC:MyORACLE";"Marc";"azerty")
If(OK=1)
 `The following query will be redirected to the external ORACLE database
   SQL EXECUTE("SELECT Name, Age FROM PERSONS";aNames;aAges)
 `The following query will be sent to the local 4D database
   Begin SQL
      SELECT Name, Age
      FROM PERSONS
      INTO :aNames, :aAges;
   End SQL
 `The following SQL LOGIN command closes the current connection
 `with the external ORACLE database and opens a new connection
 `with an external MySQL database
   SQL LOGIN("ODBC:MySQL";"Jean";"qwerty";*)
   If(OK=1)
 `The following query will be redirected to the external MySQL database
      SQL EXECUTE("SELECT Name, Age FROM PERSONS";aNames;aAges)
 `The following query will also be redirected to the external MySQL database
      Begin SQL
         SELECT Name, Age
         FROM PERSONS
         INTO :aNames, :aAges;
      End SQL
      SQL LOGOUT
 `The following query will be sent to the local 4D database
      Begin SQL
         SELECT Name, Age
         FROM PERSONS
         INTO :aNames, :aAges;
      End SQL
   End if
End if
```

## System variables and sets

If the connection is successful, the system variable OK is set to 1; otherwise, it is set to

# SQL LOGOUT

SQL LOGOUT
Does not require any parameters

## Description

The **SQL LOGOUT** command closes the connection with an ODBC source that is open in the current process (if applicable). If there is no ODBC connection open, the command does nothing.

## System variables and sets

If the logout is performed properly, the system variable OK is set to 1; otherwise, it is set to 0. You can intercept this error with an error-handling method installed by the **ON ERR CALL** command.

# SQL SET OPTION

SQL SET OPTION ( option ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ⇒ | Number of option to set |
| value | Longint, String | ⇒ | New value of option |

## Description

The **SQL SET OPTION** command modifies the *value* of the option passed in *option*.

*option* can have one of the following values, located in the "**SQL**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| SQL asynchronous | Longint | 1 | 0 = Synchronous connection (default value), 1 (or value other than 0) = Asynchronous connection |
| SQL charset | Longint | 100 | Text encoding used for requests sent to external sources (via the SQL pass-through). The modification is carried out for the current process and the current connection.<br>Possible values: MIBEnum identifier (see note 2) or value -2 (see note 3)<br>By default: 106 (UTF-8) |
| SQL connection timeout | Longint | 5 | Maximum timeout awaiting response when executing the **SQL LOGIN** command. This value must be set before opening the connection in order to be taken into account<br>Possible values: time in seconds<br>By default: no timeout |
| SQL max data length | Longint | 3 | Maximum length of data returned |
| SQL max rows | Longint | 2 | Maximum number of rows in resulting group (used for previews) |
| SQL query timeout | Longint | 4 | Maximum timeout awaiting response when executing the **SQL EXECUTE** command.<br>Values: time in seconds<br>By default: no timeout |

**Notes:**

1. When you work with the internal SQL kernel of 4D, the SQL Asynchronous option serves no purpose due to the fact that this type of connection is always synchronous.
2. MIBEnum numbers are referenced at the following address: *http://www.iana.org/assignments/character-sets*.
3. When you pass -2 as the *value* to SQL Charset, the encoding used by the 4D SQL server is automatically adapted to the running platform (non-UTF encoding):
   - Under Windows, ISO8859-1 is used,
   - Under Mac OS, MAC-ROMAN is used.

## System variables and sets

If the command was properly executed, the system variable OK returns 1. Otherwise, it returns 0.

# SQL SET PARAMETER

SQL SET PARAMETER ( object ; paramType )

| Parameter | Type | | Description |
|---|---|---|---|
| object | 4D object | ➡ | 4D object to be used (variable, array or field) |
| paramType | Longint | ➡ | Type of parameter |

## Description

The **SQL SET PARAMETER** command allows the use of a 4D variable, array or field value in SQL requests.

**Note:** It is also possible to directly insert the name of a 4D object to be used (variable, array or field) between the << and >> characters in the text of the request (see example 1). For more information about this, please refer to the **Overview of SQL Commands** section.

- In the *object* parameter, pass the 4D object (variable, array or field) to be used in the request.
- In the *paramType* parameter, pass the SQL type of the parameter. You can pass a value or use one of the following constants, located in the "**SQL**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| SQL param in | Longint | 1 | |
| SQL param in out | Longint | 2 | Usable only in the context of an SQL stored procedure (in-out parameter defined in the stored procedure) |
| SQL param out | Longint | 4 | Usable only in the context of an SQL stored procedure (out parameter defined in the stored procedure) |

The value of the 4D object replaces the *?* character in the SQL request (standard syntax). If the request contains more than one *?* character, several calls to **SQL SET PARAMETER** will be necessary. The values of the 4D objects will be assigned sequentially in the request, in accordance with the execution order of the commands.

**Warning:** This command is used for handling *parameters* passed to the SQL request. It is not possible to use the SQL param out type to associate a 4D object with the *result* of an SQL request. SQL request results are retrieved, for example, using the *boundObj* parameter of the **SQL EXECUTE** command (see the **Overview of SQL Commands**). The **SQL SET PARAMETER** command is mainly intended for setting parameters passed to the request (SQL param in); the SQL param out and SQL param in out types are reserved for use in the context of SQL stored procedures that could return parameters.

## Example 1

This example is used to execute an SQL request which calls the associated 4D variables directly:

```
C_TEXT(MyText)
C_LONGINT(MyLongint)

SQL LOGIN("mysql";"root";"")
SQLStmt:="insert into app_testTable (alpha_field, longint_field) VALUES (<<MyText>>, <<MyLongint>>)"
For(vCounter;1;10)
   MyText:="Text"+String(vCounter)
   MyLongint:=vCounter
   SQL EXECUTE(SQLStmt)
   SQL CANCEL LOAD
End for
SQL LOGOUT
```

## Example 2

Same example as the previous one, but using the **SQL SET PARAMETER** command:

```
C_TEXT(MyText)
C_LONGINT(MyLongint)
```

```
SQL LOGIN("mysql";"root";"")
SQLStmt:="insert into app_testTable (alpha_field, longint_field) VALUES (?,?)"
For(vCounter;1;10)
   MyText:="Text"+String(vCounter)
   MyLongint:=vCounter
   SQL SET PARAMETER(MyText;SQL param in)
   SQL SET PARAMETER(MyLongint;SQL param in)
   SQL EXECUTE(SQLStmt)
   SQL CANCEL LOAD
End for
SQL LOGOUT
```

## System variables and sets

If the command has been executed correctly, the system variable OK returns 1. Otherwise, it returns 0.

# START SQL SERVER

START SQL SERVER
Does not require any parameters

## Description

The **START SQL SERVER** command launches the integrated SQL server in the 4D application where it has been executed. Once launched, the SQL server can respond to external SQL queries.

**Note:** This command does not affect the internal functioning of the 4D SQL kernel. The SQL kernal is always available for internal queries.

## System variables and sets

If the SQL server has been launched correctly, the OK system variable is set to 1. Otherwise, it is set to 0.

# STOP SQL SERVER

STOP SQL SERVER
Does not require any parameters

## Description

The **STOP SQL SERVER** command stops the integrated SQL server in the 4D application where it has been executed.

If the SQL server was launched, all the SQL connections are interrupted and the server no longer accepts any external SQL queries. If the SQL server was not launched, the command does nothing.

**Note:** This command does not affect the internal functioning of the 4D SQL kernel. The SQL kernel is always available for internal queries.

# _o_USE EXTERNAL DATABASE

_o_USE EXTERNAL DATABASE ( sourceName {; user ; password} )

| Parameter | Type | | Description |
|---|---|---|---|
| sourceName | String | ⇒ | Name of ODBC data source to connect to |
| user | String | ⇒ | User name |
| password | String | ⇒ | User password |

## Compatibility Note

This command has been replaced by the **SQL LOGIN** command since version 11.3 of 4D. It should no longer be used in your developments.

# _o_USE INTERNAL DATABASE

_o_USE INTERNAL DATABASE
Does not require any parameters

## Compatibility Note

This command has been replaced by the **SQL LOGOUT** command since version 11.3 of 4D. It should no longer be used in your developments.

# String

- Character Reference Symbols
- 4D Transformation Tags
- Change string
- Char
- Character code
- CONVERT FROM TEXT
- Convert to text
- Delete string
- Get localized string
- GET TEXT KEYWORDS
- Insert string
- Length
- Lowercase
- Match regex
- Num
- Position
- Replace string
- String
- Substring
- Uppercase
- *_o_Convert case*
- *_o_ISO to Mac*
- *_o_Mac to ISO*
- *_o_Mac to Win*
- *_o_Win to Mac*

# Character Reference Symbols

## Introduction

The character reference symbols: **[[...]]**

These symbols are used to refer to a single character within a string. This syntax allows you to individually address the characters of a text variable, string variable, or field.

**Compatibility note:** Starting with 4D v13, you can no longer view the former Mac OS symbols in the Method editor (≤...≥).

If the character reference symbols appear on the left side of the assignment operator (*:=*), a character is assigned to the referenced position in the string. For example, if *vsName* is not an empty string, the following line sets the first character of *vsName* to uppercase:

```
If(vsName#"")
   vsName[[1]]:=Uppercase(vsName[[1]])
End if
```

Otherwise, if the character reference symbols appear within an expression, they return the character (to which they refer) as a 1-character string. For example:

```
  ` The following example tests if the last character of vtText is an At sign "@"
If(vtText#"")
   If(Character code(Substring(vtText;Length(vtText);1))=At sign)
  ` ...
   End if
End if

  ` Using the character reference syntax, you would write in a simpler manner:
If(vtText#"")
   If(Character code(vtText[[Length(vtText)]])=At sign)
  ` ...
   End if
End if
```

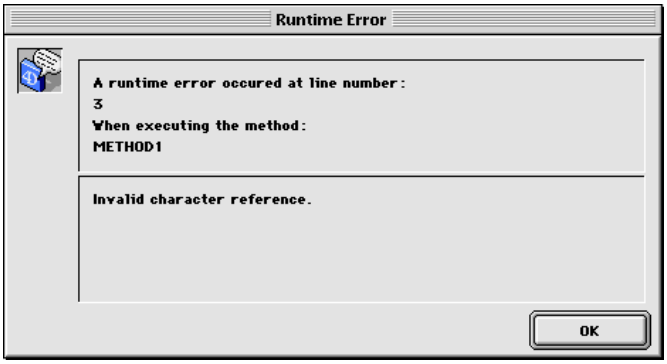
## Advanced note about invalid character reference

When you use the character reference symbols, you must address existing characters in the string in the same way you address existing elements of an array. For example if you address the 20th character of a string variable, this variable MUST contain at least 20 characters.

- Failing to do so, in interpreted mode, does not cause a syntax error.
- Failing to do so, in compiled mode (with no options), may lead to memory corruption, if, for instance, you write a character beyond the end of a string or a text.
- Failing to do so, in compiled mode, causes an error with the option Range Checking On. For example, executing the following code:

```
  ` Very bad and nasty thing to do, boo!
vsAnyText:=""
vsAnyText[[1]]:="A"
```

will trigger the Runtime Error shown here:




## Example

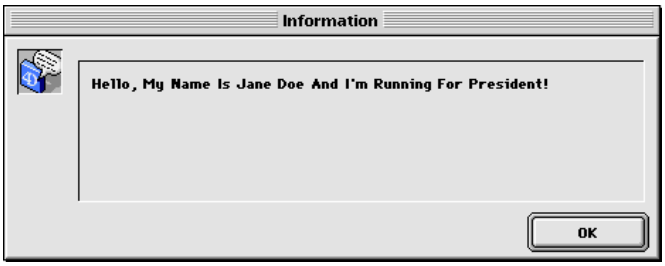
The following project method capitalizes the first character of each word of the text received as parameter and returns the resulting capitalized text:

```
  ` Capitalize text project method
  ` Capitalize text ( Text ) -> Text
  ` Capitalize text ( Source text ) -> Capitalized text

$0:=$1
$vlLen:=Length($0)
If($vlLen>0)
   $0[[1]]:=Uppercase($0[[1]])
   For($vlChar;1;$vlLen-1)
      If(Position($0[[$vlChar]];" !&()-{}:;<>?/,.=+*")>0)
         $0[[$vlChar+1]]:=Uppercase($0[[$vlChar+1]])
      End if
   End for
End if
```

For example, the line:

```
ALERT(Capitalize text("hello, my name is jane doe and i'm running for president!"))
```

displays the alert shown here:



# 4D Transformation Tags

4D provides a set of transformation tags which allow you to insert references to 4D variables or expressions, or to perform different types of processing within a source text, referred to as a "template". These tags are interpreted when the source text is executed and generate an output text.

This principle is used in particular by the 4D Web server to build the **Semi-dynamic pages**.

These tags must generally be inserted as HTML type comments (*<!--#Tag Contents-->* ) in the source text. However, other comments such as *<!--Beginning of list-->* are also possible.

**Note:** An alternative $-based syntax is used in certain conditions for tags that return values, in order to make them XML-compliant. For more information, refer to the following paragraph **Alternative syntax for 4DTEXT, 4DHTML, 4DEVAL**.

The following 4D transformation tags are available:

- 4DTEXT, to insert 4D variables and expressions as text,
- 4DHTML, to insert HTML code,
- 4DEVAL, to evaluate any 4D expression
- 4DSCRIPT, to execute a 4D method,
- 4DINCLUDE, to include a page within another one,
- 4DBASE, to modify the default folder used by the 4DINCLUDE tag,
- 4DCODE, to insert blocks of 4D code,
- 4DIF, 4DELSE, 4DELSEIF and 4DENDIF, to insert conditions in the tagged code,
- 4DLOOP and 4DENDLOOP, to make loops in the tagged code.

It is possible to mix several types of tags. For example, the following HTML structure is entirely feasible:

```
<HTML> ... <BODY> <!--#4DSCRIPT/PRE_PROCESS-->          (Method call) <!--#4DIF (myvar=1)-->                (If condition)
   <!--#4DINCLUDE banner1.html-->    (Subpage insertion) <!--#4DENDIF-->                        (End if) <!--#4DIF (mtvar=2)-->
   <!--#4DINCLUDE banner2.html--> <!--#4DENDIF--> <!--#4DLOOP [TABLE]-->                (Loop on the current selection) <!--
#4DIF ([TABLE]ValNum>10)-->          (If [TABLE]ValNum>10)    <!--#4DINCLUDE subpage.html-->   (Subpage insertion) <!--
#4DELSE-->                      (Else)    <B>Value: <!--#4DTEXT [TABLE]ValNum--></B><BR>
                              (Field display) <!--#4DENDIF--> <!--#4DENDLOOP-->                   (End for) </BODY> </HTML>
```

## Executing templates

Parsing the contents of 'template' pages can be done in two ways:

- Using the **PROCESS 4D TAGS** command; this command accepts a 'template' as input, as well as (optional) parameters and returns a text resulting from the processing.
- Using 4D's integrated HTTP server: **Semi-dynamic pages** sent by means of the **WEB SEND FILE** (.htm, .html, .shtm, .shtml), **WEB SEND BLOB** (text/html type BLOB), or **WEB SEND TEXT** commands, or called using URLs. In this last case, for reasons of optimization, pages that are suffixed with ".htm" and ".html" are NOT parsed. In order to "force" the parsing of HTML pages in this case, you must add the suffix ".shtm" or ".shtml" (for example, http://www.server.com/dir/page.shtm). For more information on this point, refer to the **Semi-dynamic pages** section in the **Web Server** chapter.

## 4DTEXT

**Syntax:** *<!--#4DTEXT VarName-->* or *<!--#4DTEXT 4DExpression-->*
**Alternative syntax:** *$4DTEXT(VarName)* or *$4DTEXT(4DExpression)* (see **Alternative syntax for 4DTEXT, 4DHTML, 4DEVAL**)

The tag *<!--#4DTEXT VarName-->* allows you to insert a reference to the 4D variable or expression returning a value. For example, if you write (in an HTML page):

<P>Welcome to *<!--#4DTEXT vtSiteName-->*!</P>

The value of the 4D variable *vtSiteName* will be inserted in the HTML page when it is sent. This value is inserted as simple text, special HTML characters such as ">" are automatically escaped.

You can also insert 4D expressions using the *4DTEXT* tag. You can for example directly insert the contents of a field (*<!--#4DTEXT [tableName]fieldName-->*), an array element (*<!--#4DTEXT tabarr{1}-->*) or a method returning a value (*<!--#4DTEXT mymethod-->*). The expression conversion follows the same rules as the variable ones. Moreover, the expression must comply with 4D syntax rules.

**Note:** Executing a 4D method with *4DTEXT* from a Web request depends on the value of the "Available through 4D tags and URLs (4DACTION...)" attribute set in the Method properties. For more information about this, refer to the **Connection Security** section.

Although an expression can contain direct calls to 4D functions, this is not recommended for localization issues. For example, *<!--#4DTEXT Current date-->*, although correctly interpreted with a 4D in English will not be understood by an French version. The same applies to real numbers (the decimal separator can be different according to the language). In both cases, we strongly advise you to assign a variable through programming.

To ensure that expressions will be evaluated correctly regardless of the 4D language or version used, we recommend using the *token* syntax for elements whose name might vary between different versions (commands, tables, fields, constants). For example, to insert the **Current time** command, enter '**Current time:C178**'. For more information about this, refer to **Using tokens in formulas**.

In case of an evaluation error, the inserted text will appear as "*<!--#4DTEXT myvar-->* : ## error # error code".

**Notes:**

- You work with process variables.
- For security reasons, it is recommended to use this tag when processing data introduced from outside the application, in order to prevent the insertion of malicious code (see the **Usage notes** section below).
- You can display a picture field content. However, it is not possible to display the content of a picture array item.
- It is possible to display the contents of an object field by means of a 4D formula. For example, you can write *<!--#4DTEXT **OB Get**:C1224([Rect]Desc;¥"color¥")-->*.
- You will usually work with Text variables. However, you can also use BLOB variables. You just need to generate the BLOB in Text without length mode.

## 4DHTML

**Syntax:** *<!--#4DHTML VarName-->* or *<!--#4DHTML 4DExpression-->*
**Alternative syntax**: *$4DHTML(VarName)* or *$4DHTML(4DExpression)* (see **Alternative syntax for 4DTEXT, 4DHTML, 4DEVAL**)

Just like the 4DTEXT tag, this tag lets you assess a variable or 4D expression that returns a value, and insert it as an HTML expression. Unlike the 4DTEXT tag, this tag does not escape HTML special characters.

For example, here are the processing results of the 4D text variable *myvar* with the available tags:

| ***myvar* Value** | **Tags** | **Result** |
|---|---|---|
| myvar:="<B>" | *<!--#4DTEXT myvar-->* | <B> |
| myvar:="<B>" | *<!--#4DHTML myvar-->* | <B> |

To ensure that expressions will be evaluated correctly regardless of the 4D language or version used, we recommend using the *token* syntax for elements whose name might vary between different versions (commands, tables, fields, constants). For example, to insert the **Current time** command, enter '**Current time:C178**'. For more information about this, refer to **Using tokens in formulas**.

In case of an interpretation error, the inserted text will be "*<!--#4DHTML myvar-->* : ## error # error code".

**Notes:**

- Executing a 4D method with *4DHTML* from a Web request depends on the value of the "Available through 4D tags and URLs (4DACTION...)" attribute set in the Method properties. For more information about this, refer to the **Connection Security** section.
- For security reasons, it is recommended to use the 4DTEXT tag when processing data introduced from outside the application, in order to prevent the insertion of malicious code (see the **Usage notes** section below).

## 4DEVAL

**Syntax:** *<!--#4DEVAL VarName--> or <!--#4DEVAL 4DExpression-->*
**Alternative syntax:** *$4DEVAL(VarName)* or *$4DEVAL(4DExpression*) (see **Alternative syntax for 4DTEXT, 4DHTML, 4DEVAL**)

The 4DEVAL tag allows you to assess a variable or a 4D expression. Like the existing 4DHTML tag, 4DEVAL does not escape HTML characters when returning text. However, unlike 4DHTML or 4DTEXT, 4DEVAL allows you to execute any valid 4D statement, including assignments and expressions that do not return any value.

For example, you can execute:

```
$input:="<!--#4DEVAL a:=42-->" //assignment
$input:=$input+"<!--#4DEVAL a+1-->" //calculation
PROCESS 4D TAGS($input;$output)
 //$output = "43"
```

To ensure that expressions will be evaluated correctly regardless of the 4D language or version used, we recommend using the *token* syntax for elements whose name might vary between different versions (commands, tables, fields, constants). For example, to insert the **Current time** command, enter '**Current time:C178**'. For more information about this, refer to **Using tokens in formulas**.

In case of an error during interpretation, the text inserted will be in the form: "*<!--#4DEVAL expr-->*: ## error # error code".

**Notes:**

- Executing a 4D method with *4DEVAL* from a Web request requires that the "Available through 4D tags and URLs (4DACTION...)" option is set in the Method properties. For more information, refer to the **Connection Security** section.
- For security reasons, it is recommended to use the 4DTEXT tag when processing data introduced from outside the application, in order to prevent the insertion of malicious code (see the **Usage notes** section below).

## 4DSCRIPT/

**Syntax:** *<!--#4DSCRIPT/MethodName/MyParam-->*

The *4DSCRIPT* tag allows you to execute 4D methods when processing the template.. The presence of the *<!--#4DSCRIPT/MyMethod/MyParam-->* tag as an HTML comment forces the execution of the ***MyMethod*** method with the *Param* parameter as a string in *$1*.

**Note:** If the tag is called in the context of a Web process, when the page is loaded, 4D calls the **On Web Authentication Database Method** (if it exists). If it returns **True**, 4D executes the method.

The method must return text in *$0*. If the string starts with the code character 1, it is considered as HTML (the same principle is true for the *4DHTML* tag).

**Note:** The execution of a method with *4DSCRIPT* depends on the value of the "Available through 4D tags and URLs (4DACTION...)" attribute defined in the Method properties. For more information about this, refer to the **Connection Security** section.

For example, let's say that you insert the following comment "Today is *<!--#4DSCRIPT/MYMETH/MYPARAM-->*" into a semi-dynamic Web page. When loading the page, 4D calls the **On Web Authentication Database Method** (if it exists), then calls the ***MYMETH*** method and passes the string "/MYPARAM" as the parameter *$1*. The method returns text in *$0* (for example "12/31/14"); the expression "Today is *<!--#4DSCRIPT/MYMETH/MYPARAM-->*" therefore becomes "Today is 12/31/14".

The MYMETH method is as follows:

```
C_TEXT($0)¥¥This parameter must always be declared
C_TEXT($1)¥¥This parameter must always be declared
$0:=String(Current date)
```

**Warning:** You must always declare the *$0* and *$1* parameters in the called method.

**Note:** A method called by *4DSCRIPT* must not call interface elements (**DIALOG**, **ALERT**...).

As 4D executes methods in their order of appearance, it is absolutely possible to call a method that sets the value of many variables that are referenced further in the document, whichever mode you are using. You can insert as many *<!--#4DSCRIPT...-->* comments as you want in a template.

## 4DINCLUDE

**Syntax:** *<!--#4DINCLUDE Path-->*

This tag is mainly designed to allow another HTML page (indicated by the *path* parameter) to be included in an HTML page. By default, only the body of the HTML page that is specified is included, in other words, the contents found within the <body> and </body> tags (the tags themselves are not included). This lets you avoid conflicts related to meta tags present in the headers. However, if the HTML page specified does not contain <body></body> tags, the entire page is included. It is

up to you to verify the consistency of the meta tags.
The *<!--#4DINCLUDE -->* comment is very useful for tests (*<!--#4DIF-->*) or loops (*<!--#4DLOOP-->*). It is very convenient to include tags according to a criteria or randomly.
When including, regardless of the file name extension, 4D analyzes the called page and then inserts the contents (modified or not) in the page originating the *4DINCLUDE* call.

An included page with the *<!--#4DINCLUDE -->* comment is loaded in the Web server cache the same way as pages called via a URL or sent with the **WEB SEND FILE** command.

In *path*, put the path leading to the document to include. **Warning**: In the case of a *4DINCLUDE* call, the path is relative to the document being analyzed, that is, the “parent” document. Use the slash character (/) as a folder separator and the two dots (..) to go up one level (HTML syntax).

**Notes:**

- When you use the *4DINCLUDE* tag with the **PROCESS 4D TAGS** command, the default folder is the folder containing the database structure file.
- You can modify the default folder used by the *4DINCLUDE* tag in the current page, using the *<!--#4DBASE -->* tag (see below).

The number of *<!--#4DINCLUDE path-->* within a page is unlimited. However, the *<!--#4DINCLUDE path-->* calls can be made only at one level. This means that, for example, you cannot insert *<!--#4DINCLUDE mydoc3.html-->* in the mydoc2.html body page, which is called by *<!--#4DINCLUDE mydoc2-->* inserted in mydoc1.html.
Furthermore, 4D verifies that inclusions are not recursive.

In case of error, the inserted text is "*<!--#4DINCLUDE path-->* :The document cannot be opened".

**Examples**

```
<!--#4DINCLUDE subpage.html--> <!--#4DINCLUDE folder/subpage.html--> <!--#4DINCLUDE ../folder/subpage.html-->
```

## 4DBASE

**Syntax:** *<!--#4DBASE folderPath-->*

The *<!--#4DBASE -->* tag designates a working directory that is used by the *<!--#4DINCLUDE-->* tag.
When it is called in a Web page, the *<!--#4DBASE -->* tag modifies all subsequent *<!--#4DINCLUDE-->* calls on this page, until the next *<!--#4DBASE -->*, if any. If the *<!--#4DBASE -->* folder is modified from within an included file, it retrieves its original value from the parent file.

The *folderPath* parameter must contain a pathname relative to the current page and it must end with a slash (/). The designated folder must be located inside the Web folder.
Pass the *WEBFOLDER* keyword to restore the default path (relative to the page).

Thus the following code (4D v12), which must specify a relative path for each call:

```
<!--#4DINCLUDE subpage.html--> <!--#4DINCLUDE folder/subpage1.html--> <!--#4DINCLUDE folder/subpage2.html--> <!--#4DINCLUDE folder/subpage3.html--> <!--#4DINCLUDE ../folder/subpage.html-->
```

... can be rewritten using the *<!--#4DBASE -->* tag:

```
<!--#4DINCLUDE subpage.html--> <!--#4DBASE folder/--> <!--#4DINCLUDE subpage1.html--> <!--#4DINCLUDE subpage2.html--> <!--#4DINCLUDE subpage3.html--> <!--#4DBASE ../folder/--> <!--#4DINCLUDE subpage.html--> <!--#4DBASE WEBFOLDER-->
```

### Example

Setting a directory for the home page using the *<!--#4DBASE -->* tag:

```
/* Index.html */ <!--#4DIF LangFR=True-->     <!--#4DBASE FR/--> <!--#4DELSE-->     <!--#4DBASE US/--> <!--#4DENDIF--> <!--#4DINCLUDE head.html--> <!--#4DINCLUDE body.html--> <!--#4DINCLUDE footer.html-->
```

In the head.html file, the current folder is modified through *<!--#4DBASE -->*, without this changing its value in Index.html:

```
/* Head.htm */ /* the working directory here is relative to the included file (FR/ or US/) */ <!--#4DBASE Styles/--> <!--#4DINCLUDE main.css--> <!--#4DINCLUDE product.css--> <!--#4DBASE Scripts/--> <!--#4DINCLUDE main.js--> <!--#4DINCLUDE product.js-->
```

## 4DCODE

The 4DCODE tag allows you to insert a multi-line 4D code block in a template.

When a "<!--#4DCODE" sequence is detected that is followed by a space, a CR or a LF character, 4D interprets all the lines of code up to the next "-->" sequence. The code block itself can contain carriage returns, line feeds, or both; it will be interpreted sequentially by 4D.

For example, using the 4DCODE tag, you can write in a template:

```
<!--#4DCODE
//PARAMETERS initialization

$graphType:=1
If(OB Is defined:C1231($graphParameters;"graphType")) //US language only
    $graphType:=OB GET:C1224($graphParameters;"graphType")
    If($graphType=7)
        $nbSeries:=1
        If($nbValues>8)
            DELETE FROM ARRAY:C228 ($yValuesArrPtr{1}->;9;100000)
            $nbValues:=8
        End if
    End if
End if
-->
```

**Note:** In a 4DCODE tag, the 4D code must always be written using the English-US language. Therefore, 4DCODE ignores the "Use regional system settings" user preferences for the 4D language (see **Language for commands and constants**).

Here are the 4DCODE tag features:

- The **TRACE** command is supported and activates the 4D debugger, thus allowing you to debug your template code.
- Any error will display the standard error dialog that lets the user stop code execution or enter debugging mode.
- The text in between <!--#4DCODE and --> is split into lines accepting any line-ending convention (cr, lf, or crlf).
- The text is tokenized within the context of the database that called **PROCESS 4D TAGS**. This is important for recognition of project methods for example.
  **Note:** The "Available through 4D tags and URLs 4DACTION" method property is not taken into account (see also the **Note about security** below).
- Even if the text always uses English-US, it is recommended to use the token syntax (:Cxxx) for command and constant names to protect against potential problems due to commands or constants being renamed from one version of 4D to another.
  **Note:** For more information on the *:Cxxx* syntax, please refer to the **Using tokens in formulas** section.

**Note about security:** The fact that 4DCODE tags can call any of the 4D language commands or project methods could be seen as a security issue, especially when the database is available through HTTP. However, since it executes server-side code called from your own template files, the tag itself does not represent a security issue. In this context, as for any Web server, security is mainly handled at the level of remote accesses to server files.

## 4DIF, 4DELSE, 4DELSEIF and 4DENDIF

**Syntax:** *<!--#4DIF expression--> {<!--#4DELSEIF expression2-->...<!--#4DELSEIF expressionN-->} {<!--#4DELSE-->} <!--#4DENDIF-->*

Used with the *<!--#4DELSEIF-->* (optional), *<!--#4DELSE-->* (optional) and *<!--#4DENDIF-->* comments, the *<!--#4DIF expression-->* comment offers the possibility to execute portions of code conditionally.

The *expression* parameter can contain any valid 4D expression returning a Boolean value. It must be indicated within parenthesis and comply with the 4D syntax rules.

To ensure that expressions will be evaluated correctly regardless of the 4D language or version used, we recommend using the *token* syntax for elements whose name might vary between different versions (commands, tables, fields, constants). For example, to insert the **Current time** command, enter '**Current time:C178**'. For more information about this, refer to **Using tokens in formulas**.

The *<!--#4DIF expression-->* ... *<!--#4DENDIF-->* blocks can be nested in several levels. Like in 4D, each *<!--#4DIF expression-->* should match a *<!--#4DENDIF-->*.

In case of an interpretation error, the text "*<!--#4DIF expression-->*: A Boolean expression was expected" is inserted instead of the contents located between *<!--#4DIF -->* and *<!--#4DENDIF-->*.

Likewise, if there are not as many *<!--#4DENDIF-->* as *<!--#4DIF -->*, the text "*<!--#4DIF expression-->*: 4DENDIF expected" is inserted instead of the contents located between *<!--#4DIF -->* and *<!--#4DENDIF-->*.

Using the *<!--#4DELSEIF-->* tag, you can test an unlimited number of conditions. Only the code that follows the first condition evaluated as **True** is executed. If no conditions are true, no statement is executed (if there is no final *<!--#4DELSE-->*).
You can use a *<!--#4DELSE-->* tag after the last *<!--#4DELSEIF-->*. If all the conditions are false, the statements following the *<!--#4DELSE-->* are executed.

The two following codes are equivalent.

- Code using 4DELSE only:

```
<!--#4DIF Condition1-->   /* Condition1 is true*/ <!--#4DELSE-->     <!--#4DIF Condition2-->         /* Condition2 is true*/     <!--#4DELSE-->
      <!--#4DIF Condition3-->            /* Condition3 is true */         <!--#4DELSE-->            /*None of the conditions are true*/
   <!--#4DENDIF-->     <!--#4DENDIF--> <!--#4DENDIF-->
```

- Similar code using the 4DELSEIF tag:

```
<!--#4DIF Condition1-->      /* Condition1 is true*/ <!--#4DELSEIF Condition2-->      /* Condition2 is true*/ <!--#4DELSEIF Condition3-->     /*
Condition3 is true */ <!--#4DELSE-->     /* None of the conditions are true*/ <!--#4DENDIF-->
```

### Example 1

This example of code inserted in a static HTML page displays a different label according the vname#"" expression result:

```
<BODY> ... <!--#4DIF (vname#"")--> Names starting with <!--#4DTEXT vname-->. <!--#4DELSE--> No name has been found. <!--
#4DENDIF--> ... </BODY>
```

### Example 2

This example inserts different pages depending on which user is connected:

```
<!--#4DIF LoggedIn=False-->     <!--#4DINCLUDE Login.htm --> <!--#4DELSEIF User="Admin" -->     <!--#4DINCLUDE
AdminPanel.htm --> <!--#4DELSEIF User="Manager" -->     <!--#4DINCLUDE SalesDashboard.htm --> <!--#4DELSE-->     <!--
#4DINCLUDE ItemList.htm --> <!--#4DENDIF-->
```

## 4DLOOP and 4DENDLOOP

**Syntax:** *<!--#4DLOOP condition--> <!--#4DENDLOOP-->*

This comment allows repetition of a portion of code as long as the condition is fulfilled. The portion is delimited by *<!--#4DLOOP-->* and *<!--#4DENDLOOP-->*.

The *<!--#4DLOOP condition-->* ... *<!--#4DENDLOOP-->* blocks can be nested. Like in 4D, each *<!--#4DLOOP condition-->* should match a *<!--#4DENDLOOP-->*.

There are five kinds of conditions:

- **<!--#4DLOOP [table]-->**
  This syntax makes a loop for each record from the *table* current selection in the current process. The code portion located between the two comments is repeated for each current selection record.

**Note:** When the 4DLOOP tag is used with a table, records are loaded in Read only mode.

The following code:

```
<!--#4DLOOP [People]--> <!--#4DTEXT [People]Name--> <!--#4DTEXT [People]Surname--><BR> <!--#4DENDLOOP-->
```

... could be expressed in 4D language in the following way:

```
FIRST RECORD([People])
While(Not(End selection([People])))
   ...
   NEXT RECORD([People])
End while
```

- **<!--#4DLOOP array-->**
  This syntax makes a loop for each item array. The array current item is increased when each code portion is repeated.

**Note:** This syntax cannot be used with two dimension arrays. In this case, it is better to combine a method with nested loops.

The following code example:

```
<!--#4DLOOP arr_names--> <!--#4DTEXT arr_names{arr_names}--><BR> <!--#4DENDLOOP-->
```

... could be expressed in 4D language in the following way:

```
For($Elem;1;Size of array(arr_names))
   arr_names:=$Elem
   ...
End for
```

- **<!--#4DLOOP method-->**
  This syntax makes a loop as long as the method returns True. The method takes a Long Integer parameter type. First it is called with the value 0 to allow an initialization stage (if necessary); it is then called with the values 1,then 2, then 3 and so on, as long as it returns True.

For security reasons, within a Web process, the **On Web Authentication database method** can be called once just before the initialization stage (method execution with 0 as parameter). If the authentication is OK, the initialization stage will proceed.

**Warning:** **C_BOOLEAN**($0) and **C_LONGINT**($1) MUST be declared within the method for compilation purposes.

The following code example:

```
<!--#4DLOOP my_method--> <!--#4DTEXT var--> <BR> <!--#4DENDLOOP-->
```

... could be expressed in 4D language in the following way:

```
If(AuthenticationWebOK)
   If(my_method(0))
      $counter:=1
      While(my_method($counter))
         ...
         $counter:=$counter+1
      End while
   End if
End if
```

The *my_method* method can be as follow:

```
C_LONGINT($1)
C_BOOLEAN($0)
If($1=0)
 `Initialisation
   $0:=True
Else
   If($1<50)
      ...
      var:=...
      $0:=True
   Else
      $0:=False `Stops the loop
   End if
End if
```

- **<!--#4DLOOP 4DExpression-->**
  With this syntax, the 4DLOOP tag makes a loop as long as the 4D expression returns **True**. The expression can be any valid Boolean expression and must contain a variable part to be evaluated in each loop to avoid infinite loops.
  For example, the following code:

```
<!--#4DEVAL $i:=0--> <!--#4DLOOP ($i<4)--> <!--#4DEVAL $i--> <!--#4DEVAL $i:=$i+1--> <!--#4DENDLOOP-->
```

produces the following result:

```
0
1
2
3
```

To ensure that expressions will be evaluated correctly regardless of the 4D language or version used, we recommend using the *token* syntax for elements whose name might vary between different versions (commands, tables, fields, constants). For example, to insert the **Current time** command, enter '**Current time:C178**'. For more information about this, refer to **Using tokens in formulas**.

- **<!--#4DLOOP pointerArray-->**
  In this case, the 4DLOOP tag works like it does with an array: it makes a loop for each element of the array referenced by the pointer. The current array element is increased each time the portion of code is repeated.
  This syntax is useful when you pass an array pointer as a parameter to the **PROCESS 4D TAGS** command.
  Example:

```
ARRAY TEXT($array;2)
$array{1}:="hello"
$array{2}:="world"
$input:="<!--#4DEVAL $1-->"
$input:=$input+"<!--#4DLOOP $2-->"
$input:=$input+"<!--#4DEVAL $2->{$2->}--> "
$input:=$input+"<!--#4DENDLOOP-->"
PROCESS 4D TAGS($input;$output;"elements = ";->$array)
 // $output = "elements = hello world "
```

In case of an interpretation error, the text "*<!--#4DLOOP expression-->*: description" is inserted instead of the contents located between *<!--#4DLOOP -->* and *<!--#4DENDLOOP-->*.

The following messages can be displayed:

- Unexpected expression type (standard error);
- Incorrect table name (error on the table name);
- An array was expected (the variable is not an array or is a two dimension array);
- The method does not exist;
- Syntax error (when the method is executing);
- Access error (you do not have the appropriate access privileges to access the table or the method).
- 4DENDLOOP expected (the *<!--#4DENDLOOP-->* number does not match the *<!--#4DLOOP -->*).

## Alternative syntax for 4DTEXT, 4DHTML, 4DEVAL

Several existing 4D transformation tags can be expressed using a $-based syntax:
**$4dtag (*expression*)** can be used instead of **<!--#4dtag *expression*-->**

This alternative syntax is available only for tags used to return processed values:

- 4DTEXT
- 4DHTML
- 4DEVAL

(Other tags, such as 4DIF or 4DSCRIPT, must be written with the regular syntax).

For example, you can write:

```
$4DEVAL(UserName)
```

instead of:

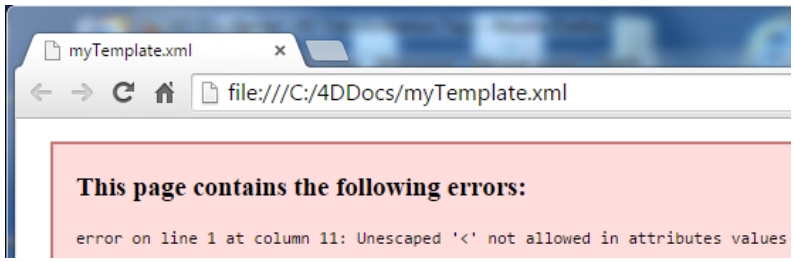
```
<!--#4DEVAL(UserName)-->
```

The main advantage of this syntax is that it allows you to **write XML-compliant templates**. Some 4D developers need to create and validate XML-based templates using standard XML parser tools. Since the "<" character is invalid in an XML

attribute value, it was not possible to use the "<!-- -->" syntax of 4D tags without breaking the document syntax. On the other hand, escaping the "<" character will prevent 4D from interpreting the tags correctly.

For example, the following code would cause an XML parsing error because of the first "<" character in the attribute value:

```
<line x1="<!--#4DEVAL $x-->" y1="<!--#4DEVAL $graphY1-->"/>
```




Using the $ syntax, the following code is validated by the parser:

```
<line x1="" y1=""/>
```

Note that *$4dtag* and *<--#4dtag -->* are not strictly equivalent: unlike *<--#4dtag -->*, *$4dtag* processing does not interpret 4D tags recursively. $ tags are always evaluated once and the result is considered as plain text.

**Note:** For more information on recursive processing, please refer to the **Recursive processing** paragraph.

The reason for this difference is to prevent malicious code injection. As explained below, it is strongly recommended to use 4DTEXT tags instead of 4DHTML tags when handling user text to protect against unwanted reinterpretation of tags: with 4DTEXT, special characters such as "<" are escaped, thus any 4D tags using the *<!--#4dtag expression -->* syntax will lose their particular meaning. However, since 4DTEXT does not escape the $ symbol, we decided to break support for recursion in order to prevent malicious injection using the *$4dtag (expression)* syntax.

The following examples show the result of processing depending on the syntax and tag used:

```
 // example 1
myName:="<!--#4DHTML QUIT 4D-->" //malicious injection
input:="My name is: <!--#4DHTML myName-->"
PROCESS 4D TAGS(input;output)
 //4D will quit!
```

```
 // example 2
myName:="<!--#4DHTML QUIT 4D-->" //malicious injection
input:="My name is: <!--#4DTEXT myName-->"
PROCESS 4D TAGS(input;output)
 //output is "My name is:  & lt;!--#4DHTML QUIT 4D-->"
```

```
 // example 3
myName:="" //malicious injection
input:="My name is: <!--#4DTEXT myName-->"
PROCESS 4D TAGS(input;output)
 //output is "My name is: ERROR: missing ')'"
```

Note that the *$4dtag* syntax supports matching pairs of enclosed quotes or parenthesis. For example, suppose that you need to evaluate the following complex (unrealistic) string:

```
String(1) + "¥"(hello)¥""
```

You can write:

```
input:="$4DEVAL( String(1)+¥"¥¥¥"(hello)¥¥¥"¥")"
PROCESS 4D TAGS(input;output)
-->output is 1"(hello)"
```

## Usage notes

### Recursive processing

4D tags are interpreted recursively: 4D always attempts to reinterpret the result of a transformation and, if a new transformation has taken place, an additional interpretation is performed, and so on until the product obtained no longer requires any further transformation. For example, given the following statement:

```
<!--#4DHTML [Mail]Letter_type-->
```

If the [Mail]Letter_type text field itself contains a tag, for example *<!--#4DSCRIPT/m_Gender-->*, this tag will be evaluated recursively after the interpretation of the 4DHTML tag.

This powerful principle meets most needs related to text transformation. Note, however, that in some cases this can also allow malicious code to be inserted. For more information about this point, refer to the following section.

### Prevention of malicious code insertion

4D transformation tags accept different types of data as parameters: text, variables, methods, command names, etc. When this data is provided by your own code, there is no risk of malicious code insertion since you control the input. However, your database code often works with data that was, at one time or another, introduced through an external source (user input, import, etc.).
In this case, it is advisable to not use transformation tags such as 4DEVAL or 4DSCRIPT, which evaluate parameters, directly with this sort of data.
In addition, according to the principle of recursion (see previous section), malicious code may itself include transformation tags. In this case, it is imperative to use the 4DTEXT tag.
Imagine, for example, a Web form field named "Name", where users must enter their name. This name is then displayed using a *<!--#4DHTML vName-->* tag in the page. If text of the "*<!--#4DEVAL **QUIT 4D**-->*" type is inserted instead of the name, interpreting this tag will cause the application to be exited.
To avoid this risk, you can just use the 4DTEXT tag systematically in this case. Since this tag escapes the special HTML characters, any malicious recursive code that may have been inserted will not be reinterpreted. To refer to the previous example, the "Name" field will contain, in this case, "*<!--#4DEVAL **QUIT 4D**-->*" which will not be transformed.

### Using the "." as decimal separator

Starting from v15 R4, 4D always uses the period character (.) as decimal separator when evaluating a numerical expression using a 4D tag (4DTEXT, 4DVAR, 4DHTML, 4DHTMLVAR, and 4DEVAL). Regional settings are now ignored.

This feature facilitates code maintenance and compatibility between 4D languages and versions.

For example, whatever the regional settings:

```
value:=10/4
input:="<!--#4DTEXT value-->"
PROCESS 4D TAGS(input;output)
 // always outputs 2.5 even if regional settings use the ',' as separator
```

**Compatibility note:** If your code evaluates numerical expressions using 4D tags with respect to the regional settings, you need to adapt it using the **String** command:

- To get *value* with a period as decimal point: *<!--#4DTEXT value-->*
- To get *value* with a decimal point based on the regional settings: *<!--#4DTEXT String(value)-->*

# Change string

Change string ( source ; newChars ; where ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| source | String | ⇒ | Original string |
| newChars | String | ⇒ | New characters |
| where | Longint | ⇒ | Where to start the changes |
| Function result | String | ⇐ | Resulting string |

## Description

**Change string** changes a group of characters in *source* and returns the resulting string. The command overlays *source*, with the characters in *newChars*, at the character described by *where*.

If *newChars* is an empty string (""), **Change string** returns *source* unchanged. **Change string** always returns a string of the same length as *source*. If *where* is less than one or greater than the length of *source*, **Change string** returns *source*.

**Change string** is different from **Insert string** in that it overwrites characters instead of inserting them.

## Example

The following example illustrates the use of **Change string**. The results are assigned to the variable *vtResult*.

```
vtResult:=Change string("Acme";"CME";2) ` vtResult gets "ACME"
vtResult:=Change string("November";"Dec";1) ` vtResult gets "December"
```

# Char

Char ( charCode ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| charCode | Longint | ⇒ | Character code |
| Function result | String | ⇐ | Character represented by the charCode |

## Description

The **Char** command returns the character whose code is *charCode*.

Pass a UTF-16 value (included between 1 and 65535) in *charCode*.

**Tip:** In editing a method, the command **Char** is commonly used to specify characters that cannot be entered from the keyboard or that would be interpreted as an editing command in the Method editor.

## Example

The following example uses **Char** to insert a carriage return within the text of an alert message:

```
ALERT("Employees: "+String(Records in table([Employees]))+Char(Carriage return)+"Press OK to continue.")
```

# Character code

Character code ( character ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| character | String | ⇒ | Character for which you want to get the code |
| Function result | Longint | ⇐ | Character code |

## Description

The **Character code** command returns the Unicode UTF-16 code (included between 1 and 65535) of *character*.

If there is more than one character in the string, **Character code** returns only the code of the first character.

The **Char** function is the counterpart of **Character code**. It returns the character that the UTF-16 code represents.

## Example 1

Uppercase and lowercase characters are considered equal within a comparison. You can use **Character code** to differentiate between uppercase and lowercase characters. Thus, this line returns True:

```
("A"="a")
```

On the other hand, this line returns False:

```
(Character code("A")=Character code("a"))
```

## Example 2

This example returns the code of the first character of the string *"ABC"*:

```
GetCode:=Character code("ABC") ` GetCode gets 65, the character code of A
```

## Example 3

The following example tests for carriage returns and tabs:

```
For($vlChar;1;Length(vtText))
   Case of
      :(vtText[$vlChar]=Char(Carriage return))
   ` Do something
      :(vtText[$vlChar]=Char(Tab))
   ` Do something else
      :(...)
   ` ...
   End case
End for
```

When executed multiple times on large texts, this test will run faster when compiled if it is written this way:

```
For($vlChar;1;Length(vtText))
   $vlCode:=Character code(vtText[$vlChar])
   Case of
      :($vlCode=Carriage return)
   ` Do something
      :($vlCode=Tab)
   ` Do something else
      :(...)
   ` ...
```

```
    End case
End for
```

The second piece of code runs faster for two reasons: it does only one character reference by iteration and uses LongInt comparisons instead of string comparisons to test for carriage returns and tabs. Use this technique when working with common codes such as CR and TAB.

# CONVERT FROM TEXT

CONVERT FROM TEXT ( 4Dtext ; charSet ; convertedBLOB )

| Parameter | Type | | Description |
|---|---|---|---|
| 4Dtext | String | ⇨ | Text expressed in current character set of 4D |
| charSet | String, Longint | ⇨ | Name or Number of character set |
| convertedBLOB | BLOB | ⇦ | BLOB containing converted text |

## Description

The **CONVERT FROM TEXT** command can be used to convert a text expressed in the current character set of 4D to a text expressed in another character set.

In the *4Dtext* parameter, pass the text to be converted. This text is expressed in the character set of 4D. In version 11, 4D uses the Unicode character set by default.

In *charSet*, pass the character set to be used for the conversion. You can pass a string containing the standard name of the set (for example “ISO-8859-1” or “UTF-8”), or its MIBEnum identifier.

Here is a list of character sets supported by the **CONVERT FROM TEXT** and **Convert to text** commands:

| MIBEnum | Name(s) |
|---|---|
| 1017 | UTF-32 |
| 1018 | UTF-32BE |
| 1019 | UTF-32LE |
| 1015 | UTF-16 |
| 1013 | UTF-16BE |
| 1014 | UTF-16LE |
| 106 | UTF-8 |
| 1012 | UTF-7 |
| 3 | US-ASCII |
| 3 | ANSI_X3.4-1968 |
| 3 | ANSI_X3.4-1986 |
| 3 | ASCII |
| 3 | cp367 |
| 3 | csASCII |
| 3 | IBM367 |
| 3 | iso-ir-6 |
| 3 | ISO_646.irv:1991 |
| 3 | ISO646-US |
| 3 | us |
| 2011 | IBM437 |
| 2011 | cp437 |
| 2011 | 437 |
| 2011 | csPC8CodePage437 |
| 2028 | ebcdic-cp-us |
| 2028 | cp037 |
| 2028 | csIBM037 |
| 2028 | ebcdic-cp-ca |
| 2028 | ebcdic-cp-n |
| 2028 | ebcdic-cp-wt |
| 2028 | IBM037 |
| 2027 | MacRoman |
| 2027 | x-mac-roman |
| 2027 | mac |
| 2027 | macintosh |
| 2027 | csMacintosh |
| 2252 | windows-1252 |
| 1250 | MacCE |
| 1250 | x-mac-ce |
| 2250 | windows-1250 |
| 1251 | x-mac-cyrillic |
| 2251 | windows-1251 |
| 1253 | x-mac-greek |
| 2253 | windows-1253 |
| 1254 | x-mac-turkish |
| 2254 | windows-1254 |
| 1256 | x-mac-arabic |
| 2256 | windows-1256 |
| 1255 | x-mac-hebrew |
| 2255 | windows-1255 |
| 1257 | x-mac-ce |
| 2257 | windows-1257 |

| | |
|---|---|
| 17 | Shift_JIS |
| 17 | csShiftJIS |
| 17 | MS_Kanji |
| 17 | Shift-JIS |
| 39 | ISO-2022-JP |
| 39 | csISO2022JP |
| 2026 | Big5 |
| 2026 | csBig5 |
| 38 | EUC-KR |
| 38 | csEUCKR |
| 2084 | KOI8-R |
| 2084 | csKOI8R |
| 4 | ISO-8859-1 |
| 4 | CP819 |
| 4 | csISOLatin1 |
| 4 | IBM819 |
| 4 | iso-ir-100 |
| 4 | ISO_8859-1 |
| 4 | ISO_8859-1:1987 |
| 4 | l1 |
| 4 | latin1 |
| 5 | ISO-8859-2 |
| 5 | csISOLatin2 |
| 5 | iso-ir-101 |
| 5 | ISO_8859-2 |
| 5 | ISO_8859-2:1987 |
| 5 | l2 |
| 5 | latin2 |
| 6 | ISO-8859-3 |
| 6 | csISOLatin3 |
| 6 | ISO-8859-3:1988 |
| 6 | iso-ir-109 |
| 6 | ISO_8859-3 |
| 6 | l3 |
| 6 | latin3 |
| 7 | ISO-8859-4 |
| 7 | csISOLatin4 |
| 7 | ISO-8859-4:1988 |
| 7 | iso-ir-110 |
| 7 | ISO_8859-4 |
| 7 | l4 |
| 7 | latin4 |
| 8 | ISO-8859-5 |
| 8 | csISOLatinCyrillic |
| 8 | cyrillic |
| 8 | ISO-8859-5:1988 |
| 8 | iso-ir-144 |
| 8 | ISO_8859-5 |
| 9 | ISO-8859-6 |
| 9 | arabic |
| 9 | ASMO-708 |
| 9 | csISOLatinArabic |

| | |
|---|---|
| 9 | ECMA-114 |
| 9 | ISO-8859-6:1987 |
| 9 | iso-ir-127 |
| 9 | ISO_8859-6 |
| 10 | ISO-8859-7 |
| 10 | csISOLatinGreek |
| 10 | ECMA-118 |
| 10 | ELOT_928 |
| 10 | greek |
| 10 | greek8 |
| 10 | iso-ir-126 |
| 10 | ISO_8859-7 |
| 10 | ISO_8859-7:1987 |
| 11 | ISO-8859-8 |
| 11 | csISOLatinHebrew |
| 11 | hebrew |
| 11 | iso-ir-138 |
| 11 | ISO_8859-8 |
| 11 | ISO_8859-8:1988 |
| 12 | ISO-8859-9 |
| 12 | csISOLatin5 |
| 12 | iso-ir-148 |
| 12 | ISO_8859-9 |
| 12 | ISO_8859-9:1989 |
| 12 | l5 |
| 12 | latin5 |
| 13 | ISO-8859-10 |
| 13 | csISOLatin6 |
| 13 | iso-ir-157 |
| 13 | ISO_8859-10 |
| 13 | ISO_8859-10:1992 |
| 13 | l6 |
| 13 | latin6 |
| 109 | ISO-8859-13 |
| 111 | ISO-8859-15 |
| 111 | Latin-9 |
| 113 | GBK |
| 2025 | GB2312 |
| 2025 | csGB2312 |
| 2025 | x-mac-chinesesimp |
| 2024 | Windows-31J |
| 57 | GB_2312-80 |
| 57 | csISO58GB231280 |

**Note:** Several rows have the same MIBEnum identifier because a character set can have more than one name (alias).

For more information about the names of character sets, please refer to the following address:
*http://www.iana.org/assignments/character-sets*

After execution of the command, the converted text will be returned in the *convertedBLOB* BLOB. This BLOB can be read by the **Convert to text** command.

## System variables and sets

If the command has been correctly executed, the OK variable is set to 1. Otherwise, it is set to 0.

# Convert to text

Convert to text ( blob ; charSet ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ➡ | BLOB containing text expressed in a specific character set |
| charSet | String, Longint | ➡ | Name or Number of BLOB character set |
| Function result | Text | ↩ | Contents of BLOB expressed in 4D character set |

## Description

The **Convert to text** command converts the text contained in the *blob* parameter and returns it in text expressed in the character set of 4D. 4D uses the UTF-16 character set by default.

In *charSet*, pass the character set of the text contained in *blob*, which will be used for the conversion. If the BLOB contains text copied from within 4D, then the BLOB's text is likely to be in the UTF-16 character set. You can pass a string providing the standard name of the character set, or one of its aliases (for example, "ISO-8859-1" or "UTF-8"), or its identifier (longint). For more information, please refer to the description of the **CONVERT FROM TEXT** command.

**Convert to text** supports Byte Order Marks (BOMs). If the character set specified is of the Unicode type (UTF-8, UTF-16 or UTF-32), 4D attempts to identify a BOM among the first bytes received. If one is detected, it is filtered out of the result and 4D uses the character set that it defines instead of the one specified.

## System variables and sets

If the command has been correctly executed, the OK variable is set to 1. Otherwise, it is set to 0.

# Delete string

Delete string ( source ; where ; numChars ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| source | String | ➡ | String from which to delete characters |
| where | Longint | ➡ | First character to delete |
| numChars | Longint | ➡ | Number of characters to delete |
| Function result | String | ↩ | Resulting string |

## Description

**Delete string** deletes *numChars* from *source*, starting at *where*, and returns the resulting string.

**Delete string** returns the same string as *source* when:

- *source* is an empty string
- *where* is greater than the length of *source*
- *numChars* is zero (0)

If *where* is less than one, the characters are deleted from the beginning of the string.

If *where* plus *numChars* is equal to or greater than the length of *source*, the characters are deleted from *where* to the end of *source*.

## Example

The following example illustrates the use of **Delete string**. The results are assigned to the variable *vtResult*.

```
vtResult:=Delete string("Lamborghini";6;6) ` vtResult gets "Lambo"
vtResult:=Delete string("Indentation";6;2) ` vtResult gets "Indention"
vtResult:=Delete string(vtOtherVar;3;32000) ` vtResult gets the first two characters of vtOtherVar
```

# Get localized string

Get localized string ( resName ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| resName | String | ⇒ | Name of resname attribute |
| Function result | String | ⇐ | Value of string designated by resName in current language |

## Description

The **Get localized string** command returns the value of the string designated by the *resName* attribute for the current language.

This command only works within an XLIFF architecture. For more information about this type of architecture, please refer to the description of XLIFF support in the *Design Reference* manual.

**Note:** The **Get database localization** command can be used to find out the language used by the application.

Pass the resource name of the string for which you want to get the translation into the current target language in *resName*.

Note that XLIFF is diacritical.

## Example

Here is an extract from an .xlf file:

```
<file source-language="en-US" target-language="fr-FR"> [...]    <trans-unit resname="Show on disk">       <source>Show on disk</source>       <target>Montrer sur le disque</target>    </trans-unit>
```

After executing the following statement:

```
$FRvalue:=Get localized string("Show on disk")
```

... if the current language is French, $FRvalue contains “Montrer sur le disque”.

## System variables and sets

If the command is executed correctly, the OK variable is set to 1. If *resName* is not found, the command returns an empty string and the OK variable is set to 0.

# GET TEXT KEYWORDS

GET TEXT KEYWORDS ( text ; arrKeywords {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| text | Text | ⇒ | Original text |
| arrKeywords | Text array | ⇐ | Array containing keywords |
| * | Operator | ⇒ | If passed = unique words |

## Description

The **GET TEXT KEYWORDS** command splits all the *text* into individual words and creates an item in the *arrKeywords* text array for each word.

4D uses the same algorithm to break up text into individual words that it does to build a **Keywords index**. It is based on the ICU library. For more information about how text is separated into words, refer to the following address: http://userguide.icu-project.org/boundaryanalysis.

**Note:** At the request of users, we introduced an exception for French and Italian: the apostrophe (') followed by either a vowel or the letter "h" is considered as a word separator. For example, the strings "L'homme" or "l'arbre" are split into "L'"+"homme" and "l'"+"arbre".

The algorithm used differs according to whether or not the **Consider only non-alphanumeric chars for keywords** option is checked in the Database settings (refer to **Database/Data storage page** in the *Design Reference* manual).

In the *text* parameter, pass the original text to be split into words. This can be styled text, in which case style tags are ignored.

For the *arrKeywords* parameter, the command fills this text array with the words extracted from the text.

If you pass the optional * parameter, the command only stores each different keyword once in *arrKeywords*. By default, if this parameter is omitted, all the words extracted from the text are stored in the array, even when they appear more than once.

This command gives you a simple way to search records containing large amounts of text with the assurance of using the same keywords as 4D. For example, imagine you have a text containing "10,000 Jean-Pierre BC45". If this text is split into the keywords "10,000" + "Jean" + "Pierre" + "BC45", then the array contains 4 elements. Then it is easy to loop through this array by programming to find records containing one or more of these keywords using the % operator (see examples).

## Example 1

In a form with a search area, users can enter one or more word(s). When a user validates this form, we look for records where the *MyField* field contains at least one of the words entered by the user.

```
 // vSearch is the variable of the search area in the form
GET TEXT KEYWORDS(vSearch;arrSearch;*)
 //* in case a user enters the same word more than once
CREATE SET([MyTable];"Totalfound")
$n:=Size of array(arrSearch)
For($i;1;$n)
   QUERY([MyTable];[MyTable]MyField % arrSearch{$i})
   CREATE SET([MyTable];"found")
   UNION("Totalfound";"found";"Totalfound")
End for
USE SET("Totalfound")
```

## Example 2

In the same form as before, we look for records where the *MyField* field contains all the words entered by the user.

```
 // vSearch is the variable of the search area in the form
GET TEXT KEYWORDS(vSearch;arrSearch;*)
$n:=Size of array(arrSearch)
QUERY([MyTable];[MyTable]MyField >=0;*)
```

```
 // initializing search = all records
For($i;1;$n)
   QUERY([MyTable];&;[MyTable]MyField % arrSearch{$i};*)
 // add criterion
End for
QUERY([MyTable]) //search
```

## Example 3

To count words in a text:

```
GET TEXT KEYWORDS(vText;arrWords) // all words
$n:=Size of array(arrWords)
GET TEXT KEYWORDS(vText;arrWords;*) // different words
$m:=Size of array(arrWords)
ALERT("This text contains "+String($n)+" separate words among "+String($m))
```

# Insert string

Insert string ( source ; what ; where ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| source | String | ➡ | String in which to insert the other string |
| what | String | ➡ | String to insert |
| where | Longint | ➡ | Where to insert |
| Function result | String | ↩ | Resulting string |

## Description

**Insert string** inserts a string into *source* and returns the resulting string. **Insert string** inserts the string *what* before the character at position *where*.

If *what* is an empty string (""), **Insert string** returns *source* unchanged.

If *where* is greater than the length of *source*, then *what* is appended to *source*. If *where* is less than one (1), then *what* is inserted before *source*.

**Insert string** is different from **Change string** in that it inserts characters instead of overwriting them.

## Example

The following example illustrates the use of **Insert string**. The results are assigned to the variable *vtResult*.

```
vtResult:=Insert string("The tree";" green";4) ` vtResult gets "The green tree"
vtResult:=Insert string("Shut";"o";3) ` vtResult gets "Shout"
vtResult:=Insert string("Indention";"ta";6) ` vtResult gets "Indentation"
```

# Length

Length ( string ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| string | String | ⇒ | String for which to return length |
| Function result | Longint | ⇐ | Length of string |

## Description

**Length** is used to find the length of *aString*. **Length** returns the number of characters that are in *aString*.

**Note:** In Unicode mode, when you want to check whether a string contains any characters, including ignorable characters, you must use the test If(Length(vtAnyText)=0) rather than If(vtAnyText=""). If the string contains for example Char(1), which is an ignorable character, Length(vtAnyText) does return 1 but vtAnyText="" returns True.

## Example

This example illustrates the use of **Length**. The results, described in the comments, are assigned to the variable *vlResult*.

```
vlResult:=Length("Topaz") ` vlResult gets 5
vlResult:=Length("Citizen") ` vlResult gets 7
```

# Lowercase

Lowercase ( aString {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aString | String | ➡ | String to convert to lowercase |
| * | Operator | ➡ | If passed: keep accents |
| Function result | String | ⮌ | String in lowercase |

## Description

**Lowercase** takes *aString* and returns the string with all alphabetic characters in lowercase.

The optional * parameter, if passed, indicates that any accented characters present in *aString* must be returned as accented lowercase characters. By default, when this parameter is omitted, accented characters “lose” their accents after the conversion is carried out.

## Example 1

The following project method capitalizes the string or text received as parameter. For instance, Caps ("john") would return "John".

```
  ` Caps project method
  ` Caps ( String ) -> String
  ` Caps ( Any text or string ) -> Capitalized text

$0:=Lowercase($1)
If(Length($0)>0)
   $0≤1≥:=Uppercase($0≤1≥)
End if
```

## Example 2

This example compares the results obtained according to whether or not the * parameter has been passed:

```
$thestring:=Lowercase("DÉJÀ VU") ` $thestring is "deja vu"
$thestring:=Lowercase("DÉJÀ VU";*) ` $thestring is "déjà vu"
```

# Match regex

Match regex ( pattern ; aString ; start {; pos_found ; length_found}{; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| pattern | Text | ⇒ | Regular expression |
| aString | Text | ⇒ | String in which search will be done |
| start | Longint | ⇒ | Position in aString where search will start |
| pos_found | Longint array, Longint variable | ⇐ | Position of occurrence |
| length_found | Longint array, Longint variable | ⇐ | Length of occurrence |
| * | Operator | ⇒ | If passed: only searches at position indicated |
| Function result | Boolean | ⇒ | True = search has found an occurrence; Otherwise, False. |

Match regex ( pattern ; aString ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| pattern | Text | ⇒ | Regular expression (complete equality) |
| aString | Text | ⇒ | String in which search will be done |
| Function result | Boolean | ⇒ | True = search has found an occurrence; Otherwise, False. |

## Description

The **Match regex** command checks the conformity of a character string with respect to a set of synthesized rules by means of a meta-language called “regular expression” or “rational expression.” The regex abbreviation is commonly used to indicate these types of notations.

Pass the regular expression to search for in *pattern*. This consists of a set of characters used for describing a character string, using special characters.

Pass the string where you want to search for the regular expression in *aString*.

In *start*, pass the position at which to start the search in *aString*.

If *pos_found* and *length_found* are variables, the command returns the position and length of the occurrence in these variables. If you pass arrays, the command returns the position and length of the occurrence in the element zero of the arrays and the positions and lengths of the groups captured by the regular expression in the following elements.

The optional * parameter indicates, when it is passed, that the search must be carried out at the position specified by *start* without searching any further in the case of failure.

The command returns **True** if the search has found an occurrence.

For more information about regex, refer to the following address:
http://en.wikipedia.org/wiki/Regular_expression

For more information about the syntax of the regular expression passed in the *pattern* parameter, refer to the following address:
http://www.icu-project.org/userguide/regexp.html

## Example 1

Search for complete equality (simple syntax):
*vfound:=Match regex(pattern;mytext)*

```
QUERY BY FORMULA([Employees];Match regex(".*smith.*";[Employees]name))
```

## Example 2

Search in text by position:
*vfound:=Match regex( pattern;mytext; start; pos_found; length_found)*
Example to display all the $1 tags:

```
$start:=1
Repeat
   vfound:=Match regex("<.*>";$1;$start;pos_found;length_found)
```

```
   If(vfound)
      ALERT(Substring($1;pos_found;length_found))
      $start:=pos_found+length_found
   End if
Until(Not(vfound))
```

## Example 3

Search with support of “capture groups” via parentheses. ( ) are used to specify groups in the regexes:
*vfound:=Match regex( pattern;mytext; start; pos_found_array; length_found_array)*

```
ARRAY LONGINT(pos_found_array;0)
ARRAY LONGINT(length_found_array;0)
vfound:=Match regex("(.*)stuff(.*)";$1;1;pos_found_array;length_found_array)
If(vfound)
   $group1:=Substring($1;pos_found_array{1};length_found_array{1})
   $group2:=Substring($1;pos_found_array{2};length_found_array{2})
End if
```

## Example 4

Search limiting the comparison of the pattern to the position indicated:
Add a star to the end of one of the two previous syntaxes.

```
vfound:=Match regex("a.b";"---a-b---";1;$pos_found;$length_found)
 `returns True
vfound:=Match regex("a.b";"---a-b---";1;$pos_found;$length_found;*)
 `returns False
vfound:=Match regex("a.b";"---a-b---";4;$pos_found;$length_found;*)
 `returns True
```

**Note:** The positions and lengths returned are only meaningful in Unicode mode or if the text being worked with is of the 7-bit ASCII type.

## Error management

In the event of an error, the command generates an error that you can intercept via a method installed by the **ON ERR CALL** command.

# Num

Num ( expression {; separator} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| expression | String, Boolean, Longint | ⇒ | String for which to return the numeric form, or Boolean to return 0 or 1, or Numeric expression |
| separator | String | ⇒ | Decimal separator |
| Function result | Longint | ⇐ | Numeric form of the expression parameter |

## Description

The **Num** command returns the numeric form of the String, Boolean or numeric expression you pass in *expression*. The optional *separator* parameter designates a decimal separator for evaluating string type expressions.

**String Expressions**

If *expression* consists only of one or more alphabetic characters, **Num** returns a zero. If *expression* includes alphabetic and numeric characters, the command ignores the alphabetic characters. Thus, it transforms the string "a1b2c3" into the number 123.

There are three reserved characters that **Num** treats specially: the decimal separator as defined in the system (if the *separator* parameter is not passed), the hyphen "-", and "*e*" *or* "E". These characters are interpreted as numeric format characters.

- The decimal separator is interpreted as a decimal place and must appear embedded in a numeric string. By default, the command uses the decimal separator set by the operating system. You can modify this character using the *separator* parameter (see below).
- The hyphen causes the number or exponent to be negative. The hyphen must appear before any negative numeric characters or after the "e" for an exponent. Except for the "e" character, if a hyphen is embedded in a numeric string, the portion of the string after the hyphen is ignored. For example, **Num**("123-456") returns 123, but **Num**("-9") returns -9.
- The e or E causes any numeric characters to its right to be interpreted as the power of an exponent. The "e" must be embedded in a numeric string. Thus, **Num**("123e-2") returns 1.23.
  Note that when the string includes more than one "e", conversion might give different results under Mac OS and under Windows.

The *separator* parameter designates a custom decimal separator for evaluating the *expression*. When the string to be evaluated is expressed with a decimal separator different from the system operator, the command returns an incorrect result. The *separator* parameter can be used in this case to obtain a correct evaluation. When this parameter is passed, the command does not take the system decimal separator into account. You can pass one or more characters.

**Note:** The **GET SYSTEM FORMAT** command can be used to find out the current decimal separator as well as several other regional system parameters.

**Boolean Expressions**

If you pass a Boolean expression, **Num** returns 1 if the expression is True; otherwise, it returns *0* (zero).

**Numeric Expressions**

If you pass a numeric expression in the *expression* parameter, **Num** returns the value passed in the *expression* parameter as is. This can be useful more particularly in the case of generic programming using pointers.

## Example 1

The following example illustrates how **Num** works when passed a string argument. Each line assigns a number to the *vResult* variable. The comments describe the results:

```
vResult:=Num("ABCD") ` vResult gets 0
vResult:=Num("A1B2C3") ` vResult gets 123
vResult:=Num("123") ` vResult gets 123
vResult:=Num("123.4") ` vResult gets 123.4
vResult:=Num("-123") ` vResult gets -123
vResult:=Num("-123e2") ` vResult gets -12300
```

## Example 2

Here, *[Client]Debt* is compared with *$1000*. The **Num** command applied to these comparisons returns 1 or 0. Multiplying 1 or 0 with a string repeats the string once or returns the empty string. As a result, *[Client]Risk* gets either “Good” or “Bad”:

```
  ` If client owes less than 1000, a good risk.
  ` If client owes more than 1000, a bad risk.
[Client]Risk:=("Good"*Num([Client]Debt<1000))+("Bad"*Num([Client]Debt>=1000))
```

## Example 3

This example compares the results obtained depending on the “current” separator:

```
$thestring:="33,333.33"
$thenum:=Num($thestring)
  ` by default, $thenum equals 33,33333 on a French system
$thenum:=Num($thestring;".")
  ` $thenum will be correctly evaluated regardless of the system;
  ` for example, 33 333,33 on a French system
```

# Position

Position ( find ; aString {; start {; lengthFound}}{; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| find | String | ⇒ | String to find |
| aString | String | ⇒ | String in which to search |
| start | Longint | ⇒ | Position in string where search will start |
| lengthFound | Longint | ⇐ | Length of string found |
| * | Operator | ⇒ | If passed: evaluation based on character codes |
| Function result | Longint | ⇒ | Position of first occurrence |

## Description

**Position** returns the position of the first occurrence of *find* in *aString*.

If *aString* does not contain *find*, it returns a zero (0).

If **Position** locates an occurrence of *find*, it returns the position of the first character of the occurrence in *aString*.

If you ask for the position of an empty string within an empty string, **Position** returns zero (0).

By default, the search begins at the first character of *aString*. The optional *start* parameter can be used to specify the character where the search will begin in *aString*.

The *lengthFound* parameter, if passed, returns the length of the string actually found by the search. This parameter is necessary to be able to correctly manage letters that can be written using one or more characters (e.g.: æ and ae, ß and ss, etc.).
Note that when the * parameter is passed (see below), these letters are not considered as equivalent (æ # ae); in this mode, *lengthFound* is always equal to the length of *find* (if an occurrence is found).

By default, the command makes global comparisons that take linguistic particularities and letters that may be written with one or more characters (for example æ = ae) into account. On the other hand, it is not diacritical (a=A, a=à and so on) and does not take "ignorable" characters into account (Unicode specification). Ignorable characters include all characters in unicode *C0 Control* subset (U+0000 to U+001F, ascii character control set) except printable ones (U+0009 TAB, U+0010 LF, U+0011 VT, U+0012 FF and U+0013 CR).
To modify this functioning, pass the asterisk * as the last parameter. In this case, comparisons will be based on character codes. You must pass the * parameter:

- If you want to take special characters into account, used for example as delimiters (**Char**(1), etc.),
- If the evaluation of characters must be case sensitive and take accented characters into account (a#A, a#à and so on). Note that in this mode, the evaluation does not handle variations in the way words are written.

**Note:** In certain cases, using the * parameter can significantly accelerate the execution of the command.

**Warning:** You cannot use the @ wildcard character with **Position**. For example, if you pass *"abc@"* in *find*, the command will actually look for *"abc@"* and not for "abc" plus any character.

## Example 1

This example illustrates the use of **Position**. The results, described in the comments, are assigned to the variable *vlResult*.

```
vlResult:=Position("ll";"Willow") ` vlResult gets 3
vlResult:=Position(vtText1;vtText2) ` Returns first occurrence of vtText1 in vtText2
vlResult:=Position("day";"Today is the first day";1) ` vlResult gets 3
vlResult:=Position("day";"Today is the first day";4) ` vlResult gets 20
vlResult:=Position("DAY";"Today is the first day";1;*) ` vlResult gets 0

vlResult:=Position("œ";"Bœuf";1;$length) ` vlResult =2, $length = 1
```

## Example 2

In the following example, the *lengthFound* parameter can be used to search for all the occurrences of "aegis" in a text, regardless of how it is written:

```
$start:=1
Repeat
   vlResult:=Position("aegis";$text;$start;$lengthfound)
   $start:=$start+$lengthfound
Until(vlResult=0)
```

# Replace string

Replace string ( source ; oldString ; newString {; howMany}{; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| source | String | ➡ | Original string |
| oldString | String | ➡ | Characters to replace |
| newString | String | ➡ | Replacement string (if empty string, occurrences are deleted) |
| howMany | Longint | ➡ | How many times to replace If omitted, all occurrences are replaced |
| * | Operator | ➡ | If passed: evaluation based on character codes |
| Function result | String | ⮌ | Resulting string |

## Description

**Replace string** replaces *howMany* occurrences of *oldString* in *source* with *newString*.

If *newString* is an empty string (""), **Replace string** deletes each occurrence of *oldString* in *source*.

If *howMany* is specified, **Replace string**will replace only the number of occurrences of *oldString* specified, starting at the first character of *source*. If *howMany* is not specified, then all occurrences of *oldString* are replaced.

If *oldString* is an empty string, **Replace string** returns the unchanged *source*.

By default, the command makes global comparisons that take linguistic particularities and letters that may be written with one or more characters (for example æ = ae) into account. On the other hand, it is not diacritical (a=A, a=à and so on) and does not take "ignorable" characters such as characters whose code < 9 into account (Unicode specification).

To modify this functioning, pass the asterisk * as the last parameter. In this case, comparisons will be based on character codes. You must pass the * parameter:

- If you want to replace special characters, used for example as delimiters (**Char**(1), etc.),
- If the replacement of characters must be case sensitive and take accented characters into account (a#A, a#à and so on).
  Note that in this mode, the evaluation does not handle variations in the way words are written.

**Note:** In 4D v15 R3 and higher, a significant optimization was made to the algorithm used by this command when you replace a string by another of a different length, regardless of the syntax used. This results in a considerable acceleration of processing in this context.

## Example 1

The following example illustrates the use of **Replace string**. The results, described in the comments, are assigned to the variable *vtResult*.

```
vtResult:=Replace string("Willow";" ll";"d") ` Result gets "Widow"
vtResult:=Replace string("Shout";"o";"") ` Result gets "Shut"
vtResult:=Replace string(vtOtherVar;Char(Tab);",";*) ` Replaces all tabs in vtOtherVar with commas
```

## Example 2

The following example eliminates CRs and TABs from the text in *vtResult*:

```
vtResult:=Replace string(Replace string(vtResult;Char(Carriage return);"";*);Char(Tab);"";*)
```

## Example 3

The following example illustrates the use of the * parameter in the case of a diacritical evaluation:

```
vtResult:=Replace string("Crème brûlée";"Brulee";"caramel") `Result gets "Crème caramel"
vtResult:=Replace string("Crème brûlée";"Brulee";"caramel";*) `Result gets "Crème brûlée"
```

# String

String ( expression {; format {; addTime}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| expression | Expression | ⇒ | Expression for which to return the string form (can be Real, Integer, Long Integer, Date, Time String, Text or Boolean) |
| format | String, Longint | ⇒ | Display format |
| addTime | Time | ⇒ | Time to add on if expression is a date |
| Function result | String | ⇐ | String form of the expression |

## Description

The **String** command returns the string form of the numeric, Date, Time, string or Boolean expression you pass in *expression*.

If you do not pass the optional *format* parameter, the string is returned with the appropriate default format. If you pass *format*, you can force the result string to be of a specific format.

The optional *addTime* parameter adds a time to a date in a combined format. It can only be used when the *expression* parameter is a date (see below).

**Numeric Expressions**

If *expression* is a numeric expression (Real, Integer, Long Integer), you can pass an optional string format. Following are some examples:

| Example | Result | Comments |
|---|---|---|
| String(2^15) | "32768" | Default format |
| String(2^15;"###,##0 Inhabitants") | "32,768 Inhabitants" | |
| String(1/3;"##0.00000") | "0.33333" | |
| String(1/3) | "0.3333333333333" | Default format(*) |
| String(Arctan(1)*4) | "3.14159265359" | Default format(*) |
| String(Arctan(1)*4;"##0.00") | "3.14" | |
| String(-1;"&x") | "0xFFFFFFFF" | |
| String(-1;"&$") | "$FFFFFFFF" | |
| String(0 ?+ 7;"&x") | "0x0080" | |
| String(0 ?+ 7;"&$") | "$80" | |
| String(0 ?+ 14;"&x") | "0x4000" | |
| String(0 ?+ 14;"&$") | "$4000" | |
| String(50.3;"&xml") | "50.3" | Always "." as decimal separator |
| String(Num(1=1);"True;;False") | "True" | |
| String(Num(1=2);"True;;False") | "False" | |
| String(Log(-1)) | "" | Undefined number |
| String(1/0) | "INF" | Positive infinite number |
| String(-1/0) | "-INF" | Negative infinite number |

(*) Beginning with 4D v14 R3, the algorithm for converting real values into text is based on 13 significant digits (as opposed to 15 digits in previous versions of 4D).

The format is specified in the same way as it would be for a number field on a form. See the section **Display formats** in the 4D Design Reference manual for more information about formatting numbers. You can also pass the name of a custom style in *format*. The custom style name must be preceded by the "|" character.

**Note:** The **String** function is not compatible with "Integer 64 bits" type fields in compiled mode.

**Date Expressions**

If *expression* is a Date expression, the string is returned using the default format specified in the system.
In the *format* parameter, you can pass one of the constants described below (**Date Display Formats** theme).

In this case, you can also pass a time in the *addTime* parameter. This parameter lets you combine a date with a time so that you can generate time stamps in compliance with current standards (ISO Date GMT and Date RFC 1123 constants). These formats are particularly useful in the context of XML and Web processing. The *addTime* parameter can only be used when the *expression* parameter is a date.

| Constant | Type | Value | Comment |
|---|---|---|---|
| Blank if null date | Longint | 100 | "" instead of 0 |
| Date RFC 1123 | Longint | 10 | |
| Internal date abbreviated | Longint | 6 | Dec 29, 2006 |
| Internal date long | Longint | 5 | December 29, 2006 |
| Internal date short | Longint | 7 | 12/29/2006 |
| Internal date short special | Longint | 4 | 12/29/06 (but 12/29/1896 or 12/29/2096) |
| ISO Date | Longint | 8 | 2006-12-29T00:00:00 (deprecated) |
| ISO Date GMT | Longint | 9 | 2010-09-13T16:11:53Z |
| System date abbreviated | Longint | 2 | Sun, Dec 29, 2006 |
| System date long | Longint | 3 | Sunday, December 29, 2006 |
| System date short | Longint | 1 | 12/29/2006 |

**Note:** Formats can vary depending on system settings.

Here are a few examples of simple formats (assuming that the current date is 12/29/2006):

```
$vsResult:=String(Current date) //$vsResult gets "12/29/06"
$vsResult:=String(Current date;Internal date long) // $vsResult gets "December 29, 2006"
$vsResult:=String(Current date;ISO Date GMT) // $vsResult gets "2009-03-04T23:00:00" in France
```

**Notes for combined date/time formats:**

- The ISO Date GMT format corresponds to the ISO8601 standard, containing a date and a time expressed with respect to the time zone (GMT).

  ```
  $mydate:=String(Current date;ISO Date GMT;Current time) // returns, for instance, 2010-09-13T16:11:53Z
  ```

  Note that the "Z" character at the end indicates the GMT format.
  If you do not pass the *addTime* parameter, the command returns the date at midnight (local time) expressed in GMT time, which may cause the date to be moved forward or back depending on the local time zone:

  ```
  $mydate:=String(!13/09/2010!;ISO Date GMT) // returns 2010-09-12T22:00:00Z in France
  ```

- The ISO Date format is similar to the ISO Date GMT, except that it expresses the date and time without respect to the time zone. Note that since this format does not comply with the ISO8601 standard, its use should be reserved for very specific purposes.

  ```
  $mydate:=String(!13/09/2010!;ISO Date) // returns 2010-09-13T00:00:00 regardless of the time zone
  $mydate:=String(Current date;ISO Date;Current time) // returns 2010-09-13T18:11:53
  ```

- The Date RFC 1123 format formats a date/time combination according to the standard defined by RFC 822 and 1123. You need this format for example to set the expiration date for cookies in an HTTP header.

  ```
  $mydate:=String(Current date;Date RFC 1123;Current time) // returns, for example Fri, 10 Sep 2010 13:07:20 GMT
  ```

  The time expressed takes the time zone into account (GMT zone). If you only pass a date, the command returns the date at midnight (local time) expressed in GMT time which may cause the date to be moved forward or back depending on the local time zone:

  ```
  $mydate:=String(Current date;Date RFC 1123) // returns Thu, 09 Sep 2010 22:00:00 GMT
  ```

**Time Expressions**

If *expression* is a Time expression, the string is returned using the default **HH:MM:SS** format. In the *format* parameter, you can pass one of the following constants (thème **Time Display Formats** theme):

| Constant | Type | Value | Comment |
|---|---|---|---|
| Blank if null time | Longint | 100 | "" instead of 0 |
| HH MM | Longint | 2 | 01:02 |
| HH MM AM PM | Longint | 5 | 1:02 AM |
| HH MM SS | Longint | 1 | 01:02:03 |
| Hour min | Longint | 4 | 1 hour 2 minutes |
| Hour min sec | Longint | 3 | 1 hour 2 minutes 3 seconds |
| ISO time | Longint | 8 | 0000-00-00T01:02:03 |
| Min sec | Longint | 7 | 62 minutes 3 seconds |
| MM SS | Longint | 6 | 62:03 |
| System time long | Longint | 11 | 1:02:03 AM HNEC (Mac only) |
| System time long abbreviated | Longint | 10 | 1•02•03 AM (Mac only) |
| System time short | Longint | 9 | 01:02:03 |

**Notes:**

- The ISO Time format corresponds to the ISO8601 standard and contains, in theory, a date and a time. Since this format does not support combined dates/times; the date part is filled with 0s. This format expresses the local time.
- The Blank if null time constant must be added to the format; it indicates that in the case of a null value, 4D must return an empty string instead of zeros.

These examples assume that the current time is 5:30 PM and 45 seconds:

```
$vsResult:=String(Current time) ` $vsResult gets "17:30:45"
$vsResult:=String(Current time;Hour Min Sec) ` $vsResult gets "17 hours 30 minutes 45 seconds"
```

**String Expressions**

If *expression* is of the String or Text type, the command returns the same value as the one passed in the parameter. This can be useful more particularly in generic programming using pointers.
In this case, the *format* parameter, if passed, is ignored.

**Boolean Expressions**

If *expression* is of the Boolean type, the command returns the string “True” or “False” in the language of the application (for example, “Vrai” or “Faux” in a French version of 4D).
In this case, the *format* parameter, if passed, is ignored.

# Substring

Substring ( source ; firstChar {; numChars} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| source | String | ➡ | String from which to get substring |
| firstChar | Longint | ➡ | Position of first character |
| numChars | Longint | ➡ | Number of characters to get |
| Function result | String | ↩ | Substring of source |

## Description

The **Substring** command returns the portion of *source* defined by *firstChar* and *numChars*.

The *firstChar* parameter points to the first character in the string to return, and *numChars* specifies how many characters to return.

If *firstChar* plus *numChars* is greater than the number of characters in the string, or if *numChars* is not specified, **Substring** returns the last character(s) in the string, starting with the character specified by *firstChar*. If *firstChar* is greater than the number of characters in the string, **Substring** returns an empty string ("").

**Warning:** When you use this command in a multi-style context, you need to convert any Window end-of-line characters ('¥r¥n') into single ('¥r') characters in order for processing to be valid. This is due to the mechanism which normalizes 4D line endings to ensure multi-platform compatibility for texts. For more information, refer to **Automatic normalization of line endings**.

## Example 1

This example illustrates the use of **Substring**. The results, described in the comments, are assigned to the variable *vsResult*.

```
vsResult:=Substring("08/04/62";4;2) ` vsResult gets "04"
vsResult:=Substring("Emergency";1;6) ` vsResult gets "Emerge"
vsResult:=Substring(var;2) ` vsResult gets all characters except ` the first
```

## Example 2

The following project method appends the paragraphs found in the text (passed as first parameter) to a string or text array (the pointer of which is passed as second parameter):

```
  ` EXTRACT PARAGRAPHS
  ` EXTRACT PARAGRAPHS ( text ; Pointer )
  ` EXTRACT PARAGRAPHS ( Text to parse ; -> Array of ¶s )

C_TEXT($1)
C_POINTER($2)

$vlElem:=Size of array($2->)
Repeat
   $vlElem:=$vlElem+1
   INSERT IN ARRAY($2->;$vlElem)
   $vlPos:=Position(Char(Carriage return);$1)
   If($vlPos>0)
      $2->{$vlElem}:=Substring($1;1;$vlPos-1)
      $1:=Substring($1;$vlPos+1)
   Else
      $2->{$vlElem}:=$1
   End if
Until($1="")
```

# Uppercase

Uppercase ( aString {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aString | String | ⇒ | String to convert to uppercase |
| * | Operator | ⇒ | If passed: keep accents |
| Function result | String | ⇐ | String in uppercase |

## Description

**Uppercase** takes *aString* and returns the string with all alphabetic characters in uppercase.

The optional * parameter, if passed, indicates that any accented characters present in *aString* must be returned as accented uppercase characters. By default, when this parameter is omitted, accented characters “lose” their accents after the conversion is carried out.

## Example 1

This example compares the results obtained according to whether or not the * parameter has been passed:

```
$thestring:=Uppercase("hélène") ` $thestring is "HELENE"
$thestring:=Uppercase("hélène";*) ` $thestring is "HÉLÈNE"
```

## Example 2

See the example for **Lowercase**.

# _o_Convert case

_o_Convert case
Does not require any parameters

## Description

*This command is obsolete and must no longer be used*.

# _o_ISO to Mac

_o_ISO to Mac ( text ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| text | String | ⇒ | Text expressed using standard Web character set |
| Function result | String | ⇐ | Text expressed using Mac OS ASCII map |

## Compatibility Note

In Unicode mode, this command does nothing (the *text* string is returned without modification). Since version 11 of 4D, this command is obsolete and its use is no longer recommended. It is recommended to convert character strings using the **CONVERT FROM TEXT** or **Convert to text** commands.

# _o_Mac to ISO

_o_Mac to ISO ( text ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| text | String | ⇒ | Text expressed using Mac OS ASCII map |
| Function result | String | ⇐ | Text expressed using standard Web character set |

## Compatibility Note

In Unicode mode, this command has no effect (the *text* string is returned without modification). Since version 11 of 4D, this command is obsolete and its use is no longer recommended. It is recommended to convert character strings using the **CONVERT FROM TEXT** or **Convert to text** commands.

# _o_Mac to Win

_o_Mac to Win ( text ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| text | String | ⇒ | Text expressed using Mac OS ASCII map |
| Function result | String | ⇐ | Text expressed using Windows ANSI map |

## Compatibility Note

In Unicode mode, this command has no effect (the *text* string is returned without modification). Since version 11 of 4D, this command is obsolete and its use is no longer recommended. It is recommended to convert character strings using the **Convert to text** or **CONVERT FROM TEXT** commands.

# _o_Win to Mac

_o_Win to Mac ( text ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| text | String | ➡ | Text expressed using Windows ANSI map |
| Function result | String | ⮌ | Text expressed using Macintosh ASCII map |

## Compatibility Note

In Unicode mode, this command has no effect (the *text* string is returned without modification). Since version 11 of 4D, this command is obsolete and its use is no longer recommended. It is recommended to convert character strings using the **Convert to text** or **CONVERT FROM TEXT** commands.

# Structure Access

- Structure Access Commands
- CREATE INDEX
- DELETE INDEX
- EXPORT STRUCTURE
- Field
- Field name
- Get external data path
- GET FIELD ENTRY PROPERTIES
- GET FIELD PROPERTIES
- Get last field number
- Get last table number
- GET MISSING TABLE NAMES
- GET RELATION PROPERTIES
- GET TABLE PROPERTIES
- IMPORT STRUCTURE
- Is field number valid
- Is table number valid
- PAUSE INDEXES
- REGENERATE MISSING TABLE
- RELOAD EXTERNAL DATA
- RESUME INDEXES
- SET EXTERNAL DATA PATH
- SET INDEX
- Table
- Table name

# Structure Access Commands

The commands in this theme return a description of the database structure. They can be used to find out the number of tables, the number of fields in each table, the names of the tables and fields, and the type and properties of each field. Utility commands can be used to detect and regenerate missing tables in order to recover "phantom" data.

Determining the database structure is extremely useful when you are developing and using groups of project methods and forms that can be copied into different databases.

The ability to read the database structure allows you to develop and use portable code.

**Note:** You can create and modify 4D fields and tables by programming using the commands of 4D's integrated SQL kernel, like **CREATE TABLE** or **ALTER TABLE**. For more information, refer to the "**4D SQL Reference**" manual.

## Counting tables and fields

It is possible to delete 4D tables and fields. You must take this possibility into account in algorithms used for counting tables and fields. It is necessary to use algorithms combining the **Get last table number** and **Get last field number**, as well as **Is table number valid** and **Is field number valid** commands. The following is an example of this type of algorithm:

```
For($thetable;1;Get last table number)
   If(Is table number valid($thetable))
      For($thefield;1;Get last field number($thetable))
         If(Is field number valid($thetable;$thefield))
            ... `The field exists and is valid
         End if
      End for
   End if
End for
```

# CREATE INDEX

CREATE INDEX ( aTable ; fieldsArray ; indexType ; indexName {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to create an index |
| fieldsArray | Pointer array | ⇒ | Pointer(s) to field(s) to be indexed |
| indexType | Longint | ⇒ | Type of index to create: -1 = Keywords, 0 = default, 1 = Standard B-Tree, 3 = Cluster B-Tree |
| indexName | Text | ⇒ | Name of index to create |
| * | Operator | ⇒ | If passed = asynchronous indexing |

## Description

The **CREATE INDEX** command creates:

- A standard index on one or more fields (composite index) or
- A keyword index on a field.

The index is created for the *aTable* table by using one or more fields designated by the *fieldsArray* pointer array. This array contains a single row when you want to create a simple index and two or more rows when you want to create a composite index (except in the case of a keyword index). In the case of composite indexes, the order of the fields in the array is important when the index is being built.

The *indexType* parameter sets the type of index to be created. You can pass one of the following constants, found in the **Index Type** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Cluster BTree index | Longint | 3 | B-Tree type index using clusters. This type of index is optimized when the index contains few keywords, i.e. when the same values occur frequently in the data. |
| Default index type | Longint | 0 | 4D specifies the index type (excluding keywords indexes) that is the most optimized according to the contents of the field. |
| Keywords index | Longint | -1 | Permits word-by-word indexing of field contents. This type of index can only be used with fields of the Text, Alpha or Picture type. Warning: Keywords indexes cannot be composite. |
| Standard BTree index | Longint | 1 | Standard B-Tree type index. This multi-purpose index type is used in previous versions of 4D |

**Note:** A B-Tree index associated with a Text type field stores the first 1024 characters of the field (maximum). Therefore in this context, searches for strings containing more than 1024 characters will fail.

In the *indexName* parameter, you pass the name of the index to be created. Naming the index is necessary if several different types of indexes can be associated with the same field and if you want to be able to delete them individually using the **DELETE INDEX** command. If the *indexName* index already exists, the command does nothing.

The optional * parameter, when it is passed, performs indexing in asynchronous mode. In this mode, the original method continues its execution after the call from the command, regardless of whether or not the indexing is finished.

If the **CREATE INDEX** command encounters any locked records, they will not be indexed and the command will wait for them to be unlocked.

If a problem occurs during command execution (non-indexed field, attempt to create a keyword index on more than one field, etc.), an error is generated. This error can be intercepted using an error-handling method.

## Example 1

Creation of two standard indexes on the “Last Name” and “Telephone”fields of the [Customers] table:

```
ARRAY POINTER(fieldPtrArr;1)
fieldPtrArr{1}:=->[Customers]LastName
CREATE INDEX([Customers];fieldPtrArr;Standard BTree Index;"CustLNameIdx")
fieldPtrArr{1}:=->[Customers]Telephone
CREATE INDEX([Customers];fieldPtrArr;Standard BTree Index;"CustTelIdx")
```

## Example 2

Creation of a keywords index on the “Observations” field of the [Customers] table:

```
ARRAY POINTER(fieldPtrArr;1)
fieldPtrArr{1}:=->[Customers]Observations
CREATE INDEX([Customers];fieldPtrArr;Keywords Index;"CustObsIdx")
```

## Example 3

Creation of a composite index on the “City” and “Zipcode” fields of the [Customers] table:

```
ARRAY POINTER(fieldPtrArr;2)
fieldPtrArr{1}:=->[Customers]City
fieldPtrArr{2}:=->[Customers]Zipcode
CREATE INDEX([Customers];fieldPtrArr;Standard BTree Index;"CityZip")
```

# DELETE INDEX

DELETE INDEX ( fieldPtr | indexName {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| fieldPtr \| indexName | Pointer, String | ⇒ | Pointer to field whose indexes are to be deleted or Name of index to be deleted |
| * | Operator | ⇒ | If passed = asynchronous operation |

## Description

The **DELETE INDEX** command deletes one or more existing indexes from the database.
You can pass either a pointer to a field or the name of an index in the parameter:

- If you pass a pointer to a field (*fieldPtr*), all the indexes associated with the field will be deleted. This can consist of keyword indexes or standard indexes. However, if the field is included in one or more composite indexes, they are not deleted (you must pass an index name).
- If you pass the name of an index (*indexName*), only the designated index will be deleted. This can consist of keyword indexes, standard indexes or composite indexes.

The optional * parameter, when it is passed, performs deindexing in asynchronous mode. In this mode, the original method continues its execution after the call from the command, regardless of whether or not the index deletion is finished.

If there is no index corresponding to *fieldPtr* or *indexName*, the command does nothing.

## Example

This example illustrates both syntaxes of the command:

```
 `Deletion of all indexes related to the LastName field
DELETE INDEX(->[Customers]LastName)
 `Deletion of index named  “CityZip”
DELETE INDEX(“CityZip”)
```

# EXPORT STRUCTURE

EXPORT STRUCTURE ( xmlStructure )

| Parameter | Type | | Description |
|---|---|---|---|
| xmlStructure | Text variable | ⇦ | Export of XML definition of 4D database structure |

## Description

The **EXPORT STRUCTURE** command exports, in *xmlStructure*, the structure definition of the current 4D database in XML format. This command uses the same mechanisms as the **Export > Structure definition to XML file...** menu item found in 4D's Design mode (see **Exporting and importing structure definitions**).

In *xmlStructure*, you pass the text variable intended to store the structure definition. The exported definition includes tables, fields, indexes, and relations, as well as their attributes and any characteristics necessary for a complete description of the structure. Invisible elements are exported with the corresponding attribute. However, deleted elements are not exported.

The internal "grammar" of 4D structure definitions is documented by means of DTD files — also used for the validation of XML files. The DTD files used by 4D are grouped together in the **DTD** folder, found next to the 4D application. The **base_core.dtd** and **common.dtd** files are used for structure definitions. For more information, you can consult these files along with the comments they contain.

## Example

You want to export the current database structure in a text file:

```
C_TEXT($vTStruc)
EXPORT STRUCTURE($vTStruc)
TEXT TO DOCUMENT("myStructure.xml";$vTStruc)
```

# Field

Field ( tableNum ; fieldNum ) -> fieldPtr

| Parameter | Type | | Description |
|---|---|---|---|
| tableNum | Longint | ⇒ | Table number |
| fieldNum | Longint | ⇒ | Field number |
| fieldPtr | Pointer | ⇐ | Field pointer |

Field ( fieldPtr ) -> fieldNum

| Parameter | Type | | Description |
|---|---|---|---|
| fieldPtr | Pointer | ⇒ | Field pointer |
| fieldNum | Longint | ⇐ | Field number |

## Description

The **Field** command has two forms:

- If you pass a table number in *tableNum* and a field number in fieldNum, **Field** returns a pointer to the field.
- If you pass a field pointer in *fieldPtr*, **Field** returns the field number of the field.

## Example 1

The following example sets the *fieldPtr* variable to a pointer to the second field in the third table:

```
FieldPtr:=Field(3;2)
```

## Example 2

Passing *fieldPtr* (a pointer to the second field of a table) to **Field** returns the number 2. The following line sets *FieldNum* to 2:

```
FieldNum:=Field(FieldPtr)
```

## Example 3

The following example sets the *FieldNum* variable to the field number of [Table3]Field2:

```
FieldNum:=Field(->[Table3]Field2)
```

# Field name

Field name ( fieldPtr | tableNum {; fieldNum} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| fieldPtr \| tableNum | Pointer, Longint | ⇒ | Field pointer or Table number |
| fieldNum | Longint | ⇒ | Field number if a table number is passed as first parameter |
| Function result | String | ⇐ | Name of the field |

## Description

The **Field name** command returns the name of the field whose pointer you pass in *fieldPtr* or whose table and field number you pass in *tableNum* and *fieldNum*.

## Example 1

This example sets the second element of the array *FieldArray*{1} to the name of the second field in the first table. *FieldArray* is a two-dimensional array:

```
FieldArray{1}{2}:=Field name(1;2)
```

## Example 2

This example sets the second element of the array *FieldArray*{1} to the name of the field *[MyTable]MyField*. *FieldArray* is a two-dimensional array:

```
FieldArray{1}{2}:=Field name(->[MyTable]MyField)
```

## Example 3

This example displays an alert. This method passes a pointer to a field:

```
ALERT("The ID number for the field "+Field name($1)+" in the table "+Table name(Table($1))+" has to be longer than five characters.")
```

# Get external data path

Get external data path ( aField ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Text, BLOB, Picture | ⇒ | Field whose storage location you want to get |
| Function result | Text | ⇐ | Full pathname of external storage file |

## Description

The **Get external data path** command returns the full pathname of the external storage file for the data of the field passed in the *aField* parameter, for the current record. You must pass fields of the Text, BLOB or Picture type in the parameter. The command returns the pathname of the storage file even if the file does not exist or is not accessible.

More particularly, this command lets you recopy the external file.

**Note:** For more information about external storage, refer to the *Design Reference* manual.

This command returns an empty string in the following cases:

- The field is not stored outside of the data file.
- The field has a Null value (and does not contain a pathname).
- The command is executed from a remote 4D.

# GET FIELD ENTRY PROPERTIES

GET FIELD ENTRY PROPERTIES ( fieldPtr|tableNum {; fieldNum}; list ; mandatory ; nonEnterable ; nonModifiable )

| Parameter | Type | | Description |
|---|---|---|---|
| fieldPtr\|tableNum | Pointer, Longint | ⇒ | Field pointer or table number |
| fieldNum | Longint | ⇒ | Field number if the table number is passed as first parameter |
| list | String | ⇐ | Associated choice list name or empty string |
| mandatory | Boolean | ⇐ | True = Mandatory, False = Optional |
| nonEnterable | Boolean | ⇐ | True = Non-enterable, False = Enterable |
| nonModifiable | Boolean | ⇐ | True = Non-modifiable, False = Modifiable |

## Description

The **GET FIELD ENTRY PROPERTIES** command returns the data entry properties for the field specified by *tableNum* and *fieldNum* or by *fieldPtr*.
You can either pass:

- table and field numbers in *tableNum* and *fieldNum*, or
- a pointer to the field in *fieldPtr*.

**Note:** This command returns the properties defined at the structure window level. Similar properties can be defined at the form level.

Once the command has been executed:

- The *list* parameter returns the choice list name associated to the field (if any). A list can be associated to the following field types: String, Text, Real, Integer, Long Integer, Date, Time and Boolean.
  If there is no choice list associated to the field or if the field type is not suitable for a choice list, an empty string is returned ("").
- The *mandatory* parameter returns True if the field is “Mandatory”; else False. The Mandatory attribute can be set for all field types, except for BLOB.
- The *nonEnterable* parameter returns True if the field is “Non-enterable”, else False. A non-enterable field can only be read, no data can be entered. The non-enterable attribute can be set for all field types, except for BLOB.
- The *nonModifiable* parameter returns True if the field is “Non-modifiable”, else False. A non-modifiable field can be entered just once and cannot be modified anymore. The Non-modifiable attribute can be set for all field types, except for BLOB.

# GET FIELD PROPERTIES

GET FIELD PROPERTIES ( fieldPtr | tableNum {; fieldNum}; fieldType {; fieldLength {; indexed {; unique {; invisible}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| fieldPtr \| tableNum | Pointer, Longint | ➡ | Table number or Field pointer |
| fieldNum | Longint | ➡ | Field number if Table number is passed |
| fieldType | Longint | ⬅ | Type of field |
| fieldLength | Longint | ⬅ | Length of field, if Alphanumeric |
| indexed | Boolean | ⬅ | True = Indexed, False = Non indexed |
| unique | Boolean | ⬅ | True = Unique, False = Non unique |
| invisible | Boolean | ⬅ | True = Invisible, False = Visible |

## Description

The **GET FIELD PROPERTIES** command returns information about the field specified by *fieldPtr* or by *tableNum* and *fieldNum*.

You either pass:

- the table and field numbers in *tableNum* and *fieldNum*, or
- a pointer to the field in *fieldPtr*.

After the call:

- *fieldType* returns the type of the field. The *fieldType* variable parameter can take a value provided by the following predefined constants (**Field and Variable Types** theme):

| Constant | Type | Value |
|---|---|---|
| Is alpha field | Longint | 0 |
| Is BLOB | Longint | 30 |
| Is Boolean | Longint | 6 |
| Is date | Longint | 4 |
| Is float | Longint | 35 |
| Is integer | Longint | 8 |
| Is integer 64 bits | Longint | 25 |
| Is longint | Longint | 9 |
| Is object | Longint | 38 |
| Is picture | Longint | 3 |
| Is real | Longint | 1 |
| Is subtable | Longint | 7 |
| Is text | Longint | 2 |
| Is time | Longint | 11 |

- The *fieldLen* parameter returns the length of the field, if the field is Alphanumeric (i.e., *fieldType*=Is alpha field). The value of *fieldLen* is meaningless for the other field types.
- The *indexed* parameter returns True is the field is indexed, and False if not. The value of *indexed* is meaningful only for Alphanumeric, Integer, Long Integer, Real, Date, Time, and Boolean fields.
- The *unique* parameter returns True if the field is set to “Unique”, else False.
- The *invisible* parameter returns True if the field is set to “Invisible”, else False. The Invisible attribute can be used to hide a given field in 4D standard editor (label, charts...).

## Example 1

This example sets the variables *vType*, *vLength*, *vIndex*, *vUnique* and *vInvisible* to the properties for the third field of the first table:

```
GET FIELD PROPERTIES(1;3;vType;vLength;vIndex;vUnique;vInvisible)
```

## Example 2

This example sets the variables *vType*, *vLength*, *vIndex*, *vUnique* and *vInvisible* to the properties for the field named [Table3]Field2:

```
GET FIELD PROPERTIES(->[Table3]Field2;vType;vLength;vIndex;vUnique;vInvisible)
```

# Get last field number

Get last field number ( tableNum | tablePtr ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| tableNum \| tablePtr | Longint, Pointer | ⇒ | Table number or Pointer to table |
| Function result | Longint | ⇐ | Highest field number in table |

## Description

The **Get last field number** command returns the highest field number among the fields in the table whose number or pointer you pass in *tableNum* or *tablePtr*.

Fields are numbered in the order in which they are created. If no field has been deleted from the table, then this command returns the number of fields that the table contains. In the case of iterative loops on the field numbers of the table, you must use the **Is field number valid** command in order to check whether the field has been deleted.

## Example

The following project method builds the array *asFields*, consisting of the field names, for the table whose pointer is received as first parameter:

```
$vlTable:=Table($1)
ARRAY STRING(31;asFields;Get last field number($vlTable))
For($vlField;Size of array(asFields);1;-1)
   If(Is field number valid($vlTable;$vlField))
      asFields{$vlField}:=Field name($vlTable;$vlField)
   Else
      DELETE FROM ARRAY(asFields;$vlField)
   End if
End for
```

# Get last table number

Get last table number -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ⮌ | Highest table number in the database |

## Description

**Get last table number** returns the highest table number among the tables in the database. Tables are numbered in the order in which they are created. If no table has been deleted from the database, this command then returns the number of tables present in the database. In the case of repeated loops on the table numbers of the database, you must use the **Is table number valid** command in order to check that the table has not been deleted.

## Example

The following example builds an array, named asTables, with the names of tables defined in the database. This array can be used as a drop-down list (or tab control, scrollable area, and so on) to display the list of the tables, within a form:

```
ARRAY STRING(31;asTables;Get last table number)
If(Get last table number>0) `If the database does contain tables
   For($vlTables;Size of array(asTables);1;-1)
      If(Is table number valid($vlTables))
         asTables{$vlTables}:=Table name($vlTables)
      Else
         DELETE FROM ARRAY(asTables;$vlTables)
      End if
   End for
End if
```

# GET MISSING TABLE NAMES

GET MISSING TABLE NAMES ( missingTables )

| Parameter | Type | | Description |
|---|---|---|---|
| missingTables | Text array | ⇦ | Names of missing tables in the database |

## Description

The **GET MISSING TABLE NAMES** command returns the names of all the missing tables of the current database in the *missingTables* array.

Missing tables are tables whose data are present in the data file but that do not exist at the level of the current structure. This can happen when a data file is opened with different versions of the structure.

Typically, the scenario is as follows:

- The developer provides a structure containing tables A, B and C,
- The user adds the custom tables D and E, using, for example, the integrated **SQL** commands of 4D, and stores data in these tables,
- The developer provides a new version of the structure. It does not contain tables D and E.
  In this case, the user version of the database still contains data from tables D and E, but it cannot be accessed. The **GET MISSING TABLE NAMES** command will return the names "D" and "E".

Once you have identified the missing tables of the database, you can reactivate them via the **REGENERATE MISSING TABLE** command.

**Note:** The data of missing tables are erased when the data file is compacted (if the tables have not been regenerated).

# GET RELATION PROPERTIES

GET RELATION PROPERTIES ( fieldPtr|tableNum {; fieldNum}; oneTable ; oneField {; choiceField {; autoOne {; autoMany}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| fieldPtr\|tableNum | Pointer, Longint | ⇒ | Field pointer or table number |
| fieldNum | Longint | ⇒ | Field number if the table number is passed as first parameter |
| oneTable | Longint | ⇐ | One table number or 0 if no relation is defined from the field |
| oneField | Longint | ⇐ | One field number or 0 if no relation is defined from the field |
| choiceField | Longint | ⇐ | Choice field number or 0 if no choice field |
| autoOne | Boolean | ⇐ | True = Auto relate one, False = Manual relate one |
| autoMany | Boolean | ⇐ | True = Auto one to many, False = Manual one to many |

## Description

The **GET RELATION PROPERTIES** command returns the properties of the relation (if any) which starts from the source field defined by *tableNum* and *fieldNum* or by *fieldPtr*.

You can pass:

- Either table and field numbers in *tableNum* and *fieldNum*,
- Or a pointer to the field in *fieldPtr*.

Once the command has been executed:

- The *oneTable* and *oneField* parameters contain respectively the table and field number to which the relation (from the source field) is pointing. If there is no relation starting from the field, these parameters return 0.
- The *choicefield* parameter contains the choice field number (from the target table) defined within this relation. If no choice field has been set for this relation, or if no relation starts from the source field, this parameter returns 0.
- The *autoOne* and *autoMany* parameters return **True** if, respectively, the “Auto Relate One” and “Auto One to Many” boxes has been checked for this relation; otherwise, they return **False**.

**Note:** The *autoOne* and *autoMany* parameters will also return **True** if no relation starts from the source field (in this case they return non-significant values). The value of both the *oneTable* and *oneField* parameters allows you to make sure that a relation exists.

# GET TABLE PROPERTIES

GET TABLE PROPERTIES ( tablePtr|tableNum ; invisible {; trigSaveNew {; trigSaveRec {; trigDelRec {; trigLoadRec}}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| tablePtr\|tableNum | Pointer, Longint | ⇒ | Table pointer or Table number |
| invisible | Boolean | ⇐ | True = Invisible, False = Visible |
| trigSaveNew | Boolean | ⇐ | True = Trigger “On saving new record” activated; otherwise, False |
| trigSaveRec | Boolean | ⇐ | True = Trigger “On saving an existing record” activated; otherwise, False |
| trigDelRec | Boolean | ⇐ | True = Trigger “On deleting a record” activated; otherwise, False |
| trigLoadRec | Boolean | ⇐ | *** Do not use (obsolete) *** |

## Description

The **GET TABLE PROPERTIES** command returns the properties for the table passed in *tablePtr* or *tableNum*. The table number or a pointer to the table can be passed as first parameter.

Once the command has been executed:

- The *invisible* parameter returns True if the “Invisible” attribute has been set for the table, else False. The Invisible attribute allows to hide the table when using 4D standard editors (label, charts...).
- The *trigSaveNew* parameter returns True if the “On saving new record” trigger has been set for the table, else False.
- The *trigSaveRec* parameter returns True if the “On saving an existing record” trigger has been set for the table, else False.
- The *trigDelRec* parameter returns True if the “On deleting a record” trigger has been set for this table, else false.

# IMPORT STRUCTURE

IMPORT STRUCTURE ( xmlStructure )

| Parameter | Type | | Description |
|---|---|---|---|
| xmlStructure | Text | ➡ | XML definition of 4D database structure |

## Description

The **IMPORT STRUCTURE** command imports, into the current database, the XML definition of the 4D database structure passed in the *xmlStructure* parameter.

The *xmlStructure* parameter must contain a valid 4D structure definition in XML format. There are several ways to obtain a valid structure definition:

- Execute the new command,
- Select the **Export > Structure definition to XML file...** menu item found in 4D's Design mode (see **Exporting and importing structure definitions**),
- Create or modify a custom XML file based on public DTDs found in the "DTD" folder of the 4D application.

The imported structure definition is added to the structure that is already open, and is displayed in the standard Structure editor of 4D among the existing tables (if any). If an imported table has the same name as a local one, an error is generated and the import operation is aborted.

You can create a new database by importing a structure definition into an empty database.

An error is generated when the structure is in compiled and/or read only mode.

A 4D application operating in remote mode cannot call this command.

## Example

You want to import a saved structure definition into the current database:

```
$struc:=Document to text("c:¥¥4DStructures¥¥Employee.xml")
IMPORT STRUCTURE($struc)
```

# Is field number valid

Is field number valid ( tableNum | tablePtr ; fieldNum ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| tableNum \| tablePtr | Longint, Pointer | ⇒ | Table number or Pointer to table |
| fieldNum | Longint | ⇒ | Field number |
| Function result | Boolean | ⇐ | True = field exists in the table False = field does not exist in the table |

## Description

The **Is field number valid** command returns True if the field whose number is passed in the *fieldNum* parameter exists in the table whose number or pointer is passed in the *tableNum* or *tablePtr* parameter. If the field does not exist, the command returns False. Keep in mind that the command returns False if the table containing the field is in the Trash of the Explorer.

This command can be used to detect any field deletions, which create gaps in the sequence of field numbers.

## Is table number valid

Is table number valid ( tableNum ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| tableNum | Longint | ⇒ | Table number |
| Function result | Boolean | ⇐ | True = table exists in database, False = table does not exist in database |

### Description

The **Is table number valid** command returns True if the table whose number is passed in the *tableNum* parameter exists in the database and False otherwise. Keep in mind that the command returns False if the table is in the Trash of the Explorer.

This command can be used to detect any table deletions, which create gaps in the sequence of table numbers.

# PAUSE INDEXES

PAUSE INDEXES ( aTable )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table for which to pause indexes |

## Description

The **PAUSE INDEXES** command temporarily disables all the indexes of *aTable*, except for the index of the primary key.

The indexes are not physically deleted from the data (.4DIndx file) or the structure of the database (_USER_INDEXES, see **System Tables**), but they are rendered invalid and are thus no longer updated. When indexes are disabled, all the operations performed on *aTable* (queries, sorts, record additions, modifications and deletions) no longer use the indexes.

This command is mainly useful when you are importing or modifying large amounts of data in tables that have several indexes. Since 4D must update the indexes each time a record is validated, the operation could take a considerable amount of time. Disabling the indexes beforehand can significantly speed up the operation.

To resume the indexes after the operation is over, you can just call the **RESUME INDEXES** command for *aTable*.

**Note:** You can obtain a similar result by using the **CREATE INDEX** and **DELETE INDEX** commands, but you will have to call them for each index of *aTable*.

If you call the **PAUSE INDEXES** command for a table and then quit the database without having called the **RESUME INDEXES** command for this table, all this table's indexes are automatically rebuilt when the database is restarted.

**Note:** This command cannot be executed from a 4D remote.

## Example

Example of method for importing large amounts of data:

```
PAUSE INDEXES([Articles])
IMPORT DATA("HugeImport.txt") //Importing
RESUME INDEXES([Articles])
```

# REGENERATE MISSING TABLE

REGENERATE MISSING TABLE ( tableName )

| Parameter | Type | | Description |
|---|---|---|---|
| tableName | Text | ➡ | Name of missing table to be regenerated |

## Description

The **REGENERATE MISSING TABLE** command rebuilds the missing table whose name is passed in the *tableName* parameter. When a missing table is rebuilt, it becomes visible in the Structure editor and its data can once again be accessed.

Missing tables are tables whose data are present in the data file but that do not exist at the structure level. You can identify any missing tables that may be present in the application by using the **GET MISSING TABLE NAMES** command.

If the table designated by the *tableName* parameter is not a missing table of the database, the command does nothing.

## Example

This method regenerates all the missing tables that may be present in the database:

```
ARRAY TEXT($arrMissingTables;0)
GET MISSING TABLE NAMES($arrMissingTables)
$SizeArray:=Size of array($arrMissingTables)
If($SizeArray#0)
 // Fills the array with the names of all the tables in the database
   ARRAY TEXT(arrTables;Get last table number)
   If(Get last table number>0) //If there are actually tables
      For($vlTables;Size of array(arrTables);1;-1)
         If(Is table number valid($vlTables))
            arrTables{$vlTables}:=Table name($vlTables)
         Else
            DELETE FROM ARRAY(arrTables;$vlTables)
         End if
      End for
   End if
   For($i;1;$SizeArray)
      If(Find in array(arrTables;$arrMissingTables{$i})=-1)
         CONFIRM("Regenerate the table"+$arrMissingTables{$i}+"?")
         If(OK=1)
            REGENERATE MISSING TABLE($arrMissingTables{$i})
         End if
      Else
         ALERT("Impossible to regenerate table "+$arrMissingTables{$i}+" because there is already a table with this name in
the database.")
      End if
   End for
 Else
   ALERT("No tables to regenerate.")
 End if
```

# RELOAD EXTERNAL DATA

RELOAD EXTERNAL DATA ( aField )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Text, BLOB, Picture, Object | ⇒ | Field for which to set the storage location |

## Description

The **RELOAD EXTERNAL DATA** command reloads the contents in memory of an external storage file associated with a BLOB, Picture, Text or Object type field.

This command is useful when the field of a record already loaded in memory is modified on the disk by another application (external storage files for fields are always writable). For example, a picture used in a Picture field can be modified by a graphic editor then saved on disk.

You then need to reload the data using the **RELOAD EXTERNAL DATA** command to update the contents of the field if it displayed in a form.

**Note:** The **RELOAD EXTERNAL DATA** command only works on a local 4D or on 4D Server. You cannot reload a field individually with 4D in remote mode. In this context, you have to reload all the records (using the **LOAD RECORD** command for example).

# RESUME INDEXES

RESUME INDEXES ( aTable {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to resume indexes |
| * | Operator | ⇒ | If passed = asynchronous indexing |

## Description

The **RESUME INDEXES** command reactivates all the indexes of *aTable* when they have been paused previously using the **PAUSE INDEXES** command. If the indexes of *aTable* have not been paused, this command does nothing.

In most cases, executing this command triggers the rebuilding of the indexes for *aTable*.
If you pass the optional * parameter, the rebuilding of the indexes is performed in asynchronous mode. This means that the method calling the command continues its execution after this call, regardless of whether the indexing is finished or not. If you omit this parameter, the rebuilding of the indexes blocks the execution of the method until the rebuilding operation is completed.

The **RESUME INDEXES** command can only be called from 4D Server or a local 4D. If this command is executed from a remote 4D machine, the error -10513 is generated. This error can be intercepted using a method installed by the **ON ERR CALL** command.

# SET EXTERNAL DATA PATH

SET EXTERNAL DATA PATH ( aField ; path )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Text, BLOB, Picture, Object | ⇒ | Field for which to set the storage location |
| path | Text, Longint | ⇒ | Pathname and file name of external storage or<br>0 = use structure definition<br>1 = use default folder |

## Description

The **SET EXTERNAL DATA PATH** command sets or modifies, for the current record, the external storage location for the *aField* field passed as parameter.

With 4D, it is possible to store Text, BLOB, Picture and Object type fields *outside* of the data file. For a complete description of this functionality, refer to the *Design Reference* manual.

The setting defined by this command is only applied when the current record is saved on the disk. If the current record is canceled, the command does nothing. Storage parameters set in the application structure are not changed.

In *path*, you can pass either a custom pathname, or a constant designating an automatic location:

- **custom pathname to file**
  In this case, you use external storage in "custom mode." Certain 4D database functions are not available automatically in this mode (see the *Design Reference* manual). In particular, you must manage the creation or modification of the files yourself.
  You can pass either a path relative to the data file or an absolute path, which must include the name and extension of the storage file. You must use the system syntax. To set a path relative to the data file, pass "../" (Windows) or "..:" (OS X) at the start of the string. You can designate any folder, including the default folder of database external files (*databaseName*.ExternalData) - in this case, these files are included when the database is saved.
  The file designated by the *path* parameter must exist and be accessible when the command is executed. Note that if the path is invalid (file or folder missing), an error is returned only in cases where the *path* was defined as absolute. When a relative *path* has been specified, you must ensure its validity since no error is generated if the file is not found.
  If you save the external file in the same folder as the data file or one of its subfolders, 4D considers that the path specified is relative to the data file and maintains the link even when the data file folder is moved or renamed.
  Note that this means it is possible to "share" the same external file between several records. Any changes made to this external file are available in all the records. In this case, if several processes can write the same fields simultaneously, you must be careful to prevent concurrent accesses through semaphores, so as not to risk damaging the external files.

- **automatic location**
  You can designate two automatic locations using the following constants, found in the **Data File Maintenance** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Use default folder | Longint | 1 | The data of the field passed as parameter are saved in the default folder, named *databaseName*.ExternalData and placed next to the data file. In this mode, external data are managed by 4D as if they were inside the data file. |
| Use structure definition | Longint | 0 | 4D uses the parameters set in the structure for field storage (see the *Design Reference* manual). If you change from external storage to internal storage, the external file is not deleted. |

Once this command is executed, 4D automatically maintains the link between the field of the record and the file on disk. You do not need to execute the command again (except if you need to change the *path*). If 4D can no longer access the data of the field (storage file renamed or deleted, path modified, etc.), the field is empty but no error is generated.

**Note:** The **SET EXTERNAL DATA PATH** command can only be executed on a local 4D or on 4D Server. It does nothing when it is executed on a remote 4D.

## Example

You want to save an existing file in the picture field, stored outside of the data, in the database folder:

```
CREATE RECORD([Photos])
[Photos]Name:="Paris.png"
SET EXTERNAL DATA PATH([Photos]Thumbnail:Get 4D folder(Database folder)+"custom"+Folder separator+[Photos]Name)
 //"/custom/Paris.png" must exist next to structure file
SAVE RECORD([Photos])
```

# SET INDEX

SET INDEX ( aField ; index {; mode} {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aField | Field | ⇒ | Field for which to create or delete the index |
| index | Boolean, Integer | ⇒ | • True=Create index, False=Delete index, or • Create an index of the type: -1=Keywords, 0=by default, 1=B-Tree standard, 3=B-Tree cluster |
| mode | Longint | ⇒ | Obsolete (parameter ignored) |
| * | | ⇒ | Asynchronous indexing if * is passed |

## Description

**Compatibility note:** This command is kept only for compatibility reasons. We now recommend using the **CREATE INDEX** and **DELETE INDEX** commands to manage indexes through programming.

The **SET INDEX** accepts two syntaxes:

- If you pass a Boolean in the*index* parameter, the command creates or removes the index for the field you pass in *aField*.
- If you pass an integer in the *index* parameter, the command creates an index of the type specified.

**index = Boolean**
To index the field, pass True in *index*. The command creates an index of the default type. If the index already exists, the call has no effect.
If you pass False in *index*, the command will delete all the standard indexes (i.e., non-composite and non-keyword) that are associated with the field. If the index does not exist, the call has no effect.

**index = Integer**
In this case, the command creates an index of the type specified for *aField*. You can pass one of the following constants, found in the "**Index Type**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Cluster BTree Index | Longint | 3 | B-Tree type index using clusters. This type of index is optimized when the index contains few keywords, i.e. when the same values occur frequently in the data. |
| Default Index Type | Longint | 0 | 4D specifies the index type (excluding keywords indexes) that is the most optimized according to the contents of the field. |
| Keywords Index | Longint | -1 | Permits word-by-word indexing of field contents. This type of index can only be used with fields of the Text or Alpha type. |
| Standard BTree Index | Longint | 1 | Standard B-Tree type index. This multi-purpose index type is used in previous versions of 4D |

**Note:** A B-Tree index associated with a Text type field stores the first 1024 characters of the field (maximum). Therefore in this context, searches for strings containing more than 1024 characters will fail.

**SET INDEX** will not index locked records; it will wait until the record becomes unlocked.

Starting with version 11, the *mode* parameter no longer serves any purpose and will be ignored if it is passed.

The optional * parameter indicates an asynchronous (simultaneous) indexing. Asynchronous indexing allows the execution of the calling method to continue immediately, whether or not indexing is completed. However, execution will halt at any command that requires the index.

**Notes:**

- Indexes created by this command do not have names. They cannot be deleted by the **DELETE INDEX** command using the syntax based on the name.
- This command cannot be used to create or delete composite indexes.
- This command cannot be used to delete a keywords index created by the **CREATE INDEX** command.

## Example 1

The following example indexes the field *[Customers]ID*:

```
UNLOAD RECORD([Customers])
SET INDEX([Customers]ID;True)
```

## Example 2

You want to index the *[Customers]Name* field in asynchronous mode:

```
SET INDEX([Customers]Name;True;*)
```

## Example 3

Creation of a keywords index:

```
SET INDEX([Books]Summary;Keywords Index)
```

# Table

Table ( tableNum | aPtr ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| tableNum \| aPtr | Longint, Pointer | ⇒ | Table number, or Table pointer, or Field pointer |
| Function result | Longint, Pointer | ⇐ | Table pointer, if a Table number is passed |
| | | | Table number, if a Table pointer is passed |
| | | | Table number, if a Field pointer is passed |

## Description

The **Table** command has three forms:

- If you pass a table number in *tableNum*, **Table** returns a pointer to the table.
- If you pass a table pointer in *aPtr*, **Table** returns the table number of the table.
- If you pass a field pointer in *aPtr*, **Table** returns the table number of the field.

## Example 1

This example sets the *tablePtr* variable to a pointer to the third table of the database:

```
TablePtr:=Table(3)
```

## Example 2

Passing *tablePtr* (a pointer to the third table) to **Table** returns the number 3. The following line sets *TableNum* to 3:

```
TableNum:=Table(TablePtr)
```

## Example 3

This example sets the *tableNum* variable to the table number of *[Table3]*:

```
TableNum:=Table(->[Table3])
```

## Example 4

This example sets the *tableNum* variable to the table number of the table to which the *[Table3]Field1* field belongs:

```
TableNum:=Table(->[Table3]Field1)
```

# Table name

Table name ( tableNum | tablePtr ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| tableNum \| tablePtr | Longint, Pointer | ⇒ | Table number or Table pointer |
| Function result | String | ⇐ | Name of the table |

## Description

The **Table name** command returns the name of the table whose number or pointer you pass in *tableNum* or *tablePtr*.

## Example

The following is an example of a generic method that displays the records of a table. The reference to the table is passed as a pointer to the table. The **Table name** command is used to include the name of the table in the title bar for the window:

```
  ` SHOW CURRENT SELECTION Project method
  ` SHOW CURRENT SELECTION ( Pointer )
  ` SHOW CURRENT SELECTION (->[Table])

SET WINDOW TITLE("Records for "+Table name($1)) `Sets the window title
DISPLAY SELECTION($1->) `Displays the selection
```

# Styled Text

The "**Styled Text**" theme contains the language commands used for managing and modifying multi-style text areas, also known as *rich text areas*.

You declare a rich text area by enabling its "Multi-style" option in the Property List (see **GET DATA SOURCE LIST** in the *Design Reference* manual). You can use the **OBJECT Is styled text** command of the "**Objects (Forms)**" theme to find out whether or not a text area is in multi-style mode.

## 4D Write Pro areas

Most commands of the "**Styled Text**" theme support 4D Write Pro areas. For more information about this point, refer to the **Using commands from the Styled Text theme** section in the 4D Write Pro Reference manual.

- Programming Notes
- Supported tags
- ST COMPUTE EXPRESSIONS
- ST FREEZE EXPRESSIONS
- ST GET ATTRIBUTES
- ST Get content type
- ST Get expression
- ST GET OPTIONS
- ST Get plain text
- ST Get text
- ST GET URL
- ST INSERT EXPRESSION
- ST INSERT URL
- ST SET ATTRIBUTES
- ST SET OPTIONS
- ST SET PLAIN TEXT
- ST SET TEXT

## Text object management commands

The commands that can be used to manipulate text objects by programming do not take any style tags integrated into the text into account. They act upon displayed text so they function as in previous versions of 4D. This concerns the following commands:

- **User Interface** theme
  **HIGHLIGHT TEXT**
  **GET HIGHLIGHT**

Note that when you use these commands with commands that manipulate character strings, it is necessary to filter the formatting characters using the **ST Get plain text** command:

```
HIGHLIGHT TEXT([Products]Notes;1;Length(ST Get plain text([Products]Notes))+1)
```

- **Objects (Forms)** theme
  The commands that can be used to modify the style of objects (for example, **OBJECT SET FONT**) apply to the whole object and not to the selection.
  Note that if the object does not have the focus when the command is executed, the modification is applied simultaneously to the object (the text area) and to its associated variable. If the object does have the focus, the modification is carried out on the object but not on the associated variable. The modification is only applied to the variable when the object loses the focus. Keep this principle in mind when programming text areas.

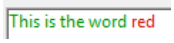
If the "Store with default style tags" option is checked for the object, the use of these commands will cause a modification of the tags saved with each object.

### Interaction of generic commands with multi-style areas

Starting with 4D v14, a new way of interacting has been defined between generic commands such as **OBJECT SET RGB COLORS** or **OBJECT SET FONT STYLE** and multi-style text areas.

In previous versions of 4D, executing one of these commands modified the contents of any custom style tags inserted in the area. From now on, only default properties are affected by these commands (as well as any properties saved by means of default tags). Custom style tags remain as they are.

For example, given a multi-style area where default tags were saved:




The plain text of the area is as follows:

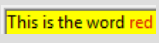
```
<span style="text-align:left;font-family:'Segoe UI';font-size:9pt;color:#009900">This is the word <span
style="color:#D81E05">red</span></span>
```

If you execute the following code:

```
OBJECT SET COLOR(*;"myArea";-(Blue+(256*Yellow)))
```

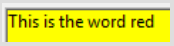
With 4D v14, the red color remains:

**4D v14 and higher**




```
<span style="text-align:left;font-
family:'Segoe UI';font-
size:9pt;color:#0000FF">This is the word
<span style="color:#D81E05">red</span>
</span>
```

**prior versions**




```
<span style="font-family:'Segoe UI';font-size:9pt;text-
align:left;font-weight:normal;font-style:normal;text-
decoration:none;color:#0000FF;"><span style="background-
color:#FFFFFF">This is the word red</span></span>
```

The following generic commands are concerned:

**OBJECT SET RGB COLORS**
**OBJECT SET COLOR**
**OBJECT SET FONT**
**OBJECT SET FONT STYLE**
**OBJECT SET FONT SIZE**

In the context of multi-style areas, generic commands should be used to set default styles only. To manage styles during database execution, we recommend using the commands of the "**Styled Text**" theme.

## Get edited text command

When it is used with a rich text area, the **Get edited text** command (**Form Events** theme) returns the text of the current area including any style tags.

To retrieve the "plain" text (text without tags) being edited, you must use the **ST Get plain text** command:

```
ST Get plain text(Get edited text)
```

## Query and order by commands

Queries and sorts carried out among multi-style objects take into account any style tags saved in the object. If a style modification has been made within a word, searching for the word will not be successful.

To be able to carry out valid searches and sorts, you must use the **ST Get plain text** command. For example:

```
QUERY BY FORMULA([MyTable];ST Get plain text([MyTable]MyFieldStyle)="very well")
```

## Automatic normalization of line endings

In order to ensure greater multi-platform compatibility of texts handled in the database, beginning with v14, 4D automatically normalizes line endings so that they occupy a single character: '¥r' (carriage return). This normalization is carried out at the level of form objects (variables or fields) hosting plain or multi-style text. Line endings that are not native, or that use a mix of several characters (for example '¥r¥n'), are considered as a single '¥r'.

Note that in compliance with the XML standard (multi-style text format), the multi-style text commands also normalize line endings for text variables that are not associated with objects.

This principle makes it easier to use multi-style text commands or commands such as **HIGHLIGHT TEXT** in a multi-platform context. However, you must take this into account in your processing when you work with texts from heterogeneous sources.

## Supported tags

### Dynamic tags

You can use the following tags in 4D multi-style text areas.

#### 4D Expression

This tag inserts a 4D expression (expression, method, field, variable, command, etc.) in the text. The expression is tokenized and evaluated:

- when the expression is inserted
- when the object is loaded
- when the **ST COMPUTE EXPRESSIONS** command is executed
- when the **ST FREEZE EXPRESSIONS** command is executed, if the second * parameter is passed.

The evaluated value of the expression is not saved in the  tag, only its reference is.

**Note:** To ensure that expressions will be evaluated correctly regardless of the 4D language or version used, we recommend using the *token* syntax for elements whose name might vary between different versions (commands, tables, fields, constants). For example, to insert the **Current time** command, enter '**Current time:C178**'. For more information about this, refer to **Using tokens in formulas**.

#### URL

<a href="*url*">Visible label</a>

This tag inserts a URL in the text. Example:

```
<span><a href="http://www.4d.com/">4D Web Site</a></span>
```

#### User link

Click here

"User links" look the same as URLs, but when you click them, they do not automatically open the source. You can pass any string you want as reference, and it is up to the developer to program any custom actions that occur when it is clicked. This means you can create links which are not URLs but references to files, 4D methods, and so on, that you can open or execute when they are clicked. The **ST Get content type** command detects if a user link has been clicked.

User links are defined using the **ST SET TEXT** command. For example:

```
ST SET TEXT(txtVar;"This is a user link: <span style=¥"-d4-ref-user:'UserLink'¥">User Label</span>";$start;$end)
```

#### Custom tags

You can now insert any tag in plain text, for example <img src="http://doc.4d.com/pictures/ja.png">. It is stored in the code of the plain text without being interpreted or displayed. This is particularly useful in the context of e-mails in HTML format and including pictures for example.

### Style tags

This paragraph lists the attributes of <SPAN> tags that are supported by 4D in rich text areas. You can use these tags to implement custom style handling. Only the tags listed below are supported by 4D for style variations.

#### Font name

<SPAN STYLE="font-family: DESDEMONA"> ... </SPAN>

#### Font size

<SPAN STYLE="font-size: 20pt"> ... </SPAN>

#### Font style

- **Bold**
  <SPAN STYLE="font-weight: bold"> ... </SPAN>
- **Italic or normal**
  <SPAN STYLE="font-style: italic"> ... </SPAN>
  <SPAN STYLE="font-style: normal"> ... </SPAN>
- **Underline**
  <SPAN STYLE="text-decoration: underline"> ... </SPAN>
- **Strikethrough**
  <SPAN STYLE="text-decoration:line-through">...</SPAN>
  **Note :** The "strikethrough" style is not supported under Mac OS, but this tag can still be managed by programming.

### Font colors

<SPAN STYLE="color:green"> ... </SPAN>
or
<SPAN STYLE="color:#006CCC">...</SPAN>

### Background colors (Windows only)

<SPAN STYLE="background-color:green"> ... </SPAN>
or
<SPAN STYLE="background-color:#006CCC">...</SPAN>

**Note:** Under Mac OS, this attribute is ignored. It is removed when the object is modified.

### Color values

For font color and background color attributes, the color value can be either the hexadecimal code for an RGB color, or the name of one of the 16 HTML colors defined for standard CSS by the W3C:

| Color | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | Aqua | Black | Blue | Fuchsia | Gray | Green | Lime | Maroon |
| RGB | #00FFFF | #000000 | #0000FF | #FF00FF | #808080 | #008000 | #00FF00 | #800000 |
| Color | | | | | | | | |
| Name | Navy | Olive | Purple | Red | Silver | Teal | White | Yellow |
| RGB | #000080 | #808000 | #800080 | #FF0000 | #C0C0C0 | #008080 | #FFFFFF | #FFFF00 |

# ST COMPUTE EXPRESSIONS

ST COMPUTE EXPRESSIONS ( {* ;} object {; startSel {; endSel}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or<br>Field or variable (if * is omitted) |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |

## Description

The **ST COMPUTE EXPRESSIONS** command updates the dynamic 4D expressions found in the styled text field or variable designated by the *object* parameter.

For more information about 4D expressions used in multi-style text areas, refer to the description of the **ST INSERT EXPRESSION** command.

The command re-evaluates the result of expressions found in the *object* based on the current context and displays the result obtained. For example, if the expression inserted is the time, the value will be modified each time the **ST COMPUTE EXPRESSIONS** command is called. Expressions are also computed:

- when they are inserted
- when the object is loaded
- when they are "frozen" using the **ST FREEZE EXPRESSIONS** command, if the second * parameter is passed.

**ST COMPUTE EXPRESSIONS** does not modify styled text (containing *span* tags) but only plain text displayed in *object*. The values computed are not stored in the styled text, only their reference is stored there.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

It is not necessary for the *object* to have the focus; however, the object must be included in a form, or else the **ST COMPUTE EXPRESSIONS** command has no effect.

The optional *startSel* and *endSel* parameters designate a selection of text in *object*. The *startSel* and *endSel* values express a plain text selection, without taking into account any style tags or references that may be present. Note that a reference is equivalent to a single character.

- If you pass *startSel* and *endSel*, **ST COMPUTE EXPRESSIONS** only updates the expressions located within this selection.
- If you only pass *startSel* or if the value of *endSel*is greater than the total number of characters in object, all the expressions between *startSel* and the end of the text are computed.
- If you omit *startSel* and *endSel*, all the expressions included in the user selection of the *object* are computed.

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

**Note:** If *startSel* is greater than *endSel* (except when *endSel* is 0), the command does nothing and the *OK* variable is set to 0.

## Example

You want to update the references included in the selection of text:

```
ST COMPUTE EXPRESSIONS(*;"myText";ST Start highlight;ST End highlight)
```

# ST FREEZE EXPRESSIONS

ST FREEZE EXPRESSIONS ( {* ;} object {; startSel {; endSel}}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |
| * | Operator | ⇒ | If passed = update expressions before freezing them |

## Description

The **ST FREEZE EXPRESSIONS** command "freezes" the contents of expressions found in the styled text field or variable designated by the *object* parameter. This action converts dynamic expressions into static text and removes the associated references from the *object*.

For more information about 4D expressions used in multi-style text areas, refer to the description of the **ST INSERT EXPRESSION** command.

The **ST FREEZE EXPRESSIONS** command stores the computed value of an expression at a given time. This operation is necessary particularly before each use of the *object* outside of a multi-style area (exports, storage in a disk file, printing, etc.) since only the reference of the expression is kept in the area itself.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

The optional *startSel* and *endSel* parameters designate a selection of text in *object*. The *startSel* and *endSel* values express a plain text selection, without taking into account any style tags or references that may be present.

- If you pass *startSel* and *endSel*, **ST FREEZE EXPRESSIONS** only freezes the expressions located within this selection.
- If you only pass *startSel* or if the value of *endSel* is greater than the total number of characters in the *object*, all the expressions between *startSel* and the end of the text are frozen.
- If you omit *startSel* and *endSel*, all the expressions included in the user selection of the *object* are frozen.

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters.
These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

**Note:** If *startSel* is greater than *endSel* (except when *endSel*is 0), the command does nothing and the *OK* variable is set to 0.

By default, expressions are not re-evaluated before they are frozen. If you want the expression to be recomputed and then frozen, you can pass the second * parameter.

## Example

You want to insert the current time at the start of the text and then freeze it before saving the record:

```
 //Inserting the time at the start of the text
ST INSERT EXPRESSION(*;StyledText_t;"Current time";1)
 //We freeze the expression
ST FREEZE EXPRESSIONS(*;"StyledText_t";1)
```

# ST GET ATTRIBUTES

ST GET ATTRIBUTES ( {* ;} object ; startSel ; endSel ; attribName ; attribValue {; attribName2 ; attribValue2 ; ... ; attribNameN ; attribValueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| startSel | Longint | ⇒ | Start of text selection |
| endSel | Longint | ⇒ | End of text selection |
| attribName | Longint | ⇒ | Attribute to get |
| attribValue | Variable | ⇐ | Current value of attribute |

## Description

The **ST GET ATTRIBUTES** command is used to recover the current value of a style attribute in a selection of text of the form object(s) designated by *object*.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (string). During execution, if the object has the focus, the command returns information about the object being edited; however, when the object does not have the focus, the command returns information about the data source (field or variable) of the object.
If you omit the * parameter, this indicates that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference instead of a string and during execution, the command returns information about this field or variable.

The *startSel* and *endSel* parameters are used to designate the text selection of the object from which the style attribute is to be read. Pass the position of the first character of the selection in *startSel* and the position plus one of the last character of the selection in *endSel*. You can pass 0 in *endSel* to designate automatically the last character of the text (pass 1 in *startSel* to designate the first character of the text).
If the values of *startSel* and *endSel* are equal or if *startSel* is greater than *endSel* (except if *endSel* value is 0, see above), an error is returned.
The *startSel* and *endSel* values do not take any style tags already present in the area into account. They are evaluated on the basis of raw text (text from which style tags have been filtered).

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

Pass the name of the attribute to get in the *attribName* parameter and pass a variable which must recover the current value of the attribute in the *attribValue* parameter. To specify the *attribName* parameter, you must use one of the constants of the **Multistyle Text Attributes** theme.

| Constant | Type | Value | Comment |
|---|---|---|---|
| Attribute background color | Longint | 8 | *attribValue*=Hexadecimal values or HTML color names (Windows only) |
| Attribute bold style | Longint | 1 | *attribValue*=0: remove bold attribute from selection<br>*attribValue*=1: apply bold attribute to selection |
| Attribute font name | Longint | 5 | *attribValue*=Font family name (string) |
| Attribute italic style | Longint | 2 | *attribValue*=0: remove italic attribute from selection<br>*attribValue*=1: apply italic attribute to selection |
| Attribute strikethrough style | Longint | 3 | *attribValue*=0: remove strikethrough attribute from selection<br>*attribValue*=1: apply strikethrough attribute to selection |
| Attribute text color | Longint | 7 | *attribValue*=Hexadecimal values or HTML color names |
| Attribute text size | Longint | 6 | *attribValue*=Number of points (number) |
| Attribute underline style | Longint | 4 | *attribValue*=0: remove underline attribute from selection<br>*attribValue*=1: apply underline attribute to selection |

You can pass as many attribute/value pairs as you want.

If the value of the *attribName* attribute is the same for all of the selection, it is returned in *attribValue*. If this value is different or if *object* does not contain SPAN tags, the following values are returned:

| attribName | attribValue if attribute heterogenous in selection or no SPAN tags |
|---|---|
| Attribute background color | FFFFFFFF |
| Attribute bold style | 2 |
| Attribute font name | "" (empty string) |
| Attribute italic style | 2 |
| Attribute strikethrough style | 2 |
| Attribute text color | FFFFFFFF |
| Attribute text size | -1 |
| Attribute underline style | 2 |

## Example

Given a [Table_1]StyledText field displayed in a form. The object has the Multistyle property and is named "StyledText_t". You want to get the highlighted text as well as the status of the Bold style attribute. You can proceed in two different ways depending on whether you use the object name or the field reference.

- Using the object name:

```
$text:=ST Get text(*;"StyledText_t";ST Start highlight;ST End highlight)
ST GET ATTRIBUTES(*;"StyledText_t";ST Start highlight;ST End highlight;Attribute bold style;$bold)
```

- Using the field name:

```
GET HIGHLIGHT([Table_1]StyledText;$Begin_l;$End_l)
$text:=ST Get text([Table_1]StyledText;$Begin_l;$End_l)
ST GET ATTRIBUTES([Table_1]StyledText;$Begin_l;$End_l;Attribute bold style;$bold)
```

## System variables and sets

After this command is executed, the OK variable is set to 1 if no error occurred; otherwise, it is set to 0. This is the case more particularly when style tags are not evaluated properly (incorrect or missing tags).

In the case of an error, the variable is not changed. When an error occurs on a variable when text is being evaluated, 4D transforms the text into plain text; as a result, the <, > and & characters are converted into HTML entities.

# ST Get content type

ST Get content type ( {* ;} object {; startSel {; endSel {; startBlock {; endBlock}}}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |
| startBlock | Longint | ⇐ | Start position of first type of selection |
| endBlock | Longint | ⇐ | End position of first type of selection |
| Function result | Longint | ⇐ | Type of content |

## Description

The **ST Get content type** command returns the type of content found in the styled text field or variable designated by the *object* parameter.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). During execution, if the object has the focus, the command returns the information of the object being edited; if the object does not have the focus, the command returns the information of the object's data source (variable or field).
If you omit the * parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string. During execution, the command returns the information of the variable or field.

The optional *startSel* and *endSel* parameters designate a selection of text in *object*. The *startSel* and *endSel* values express a plain text selection, without taking into account any style tags that may be present.

- If you pass *startSel* and *endSel*, **ST Get content type** evaluates the contents within this selection.
- If you only pass *startSel* or if the value of *endSel*is greater than the total number of characters in *object*, the contents between *startSel* and the end of the text is evaluated.
- If you omit *startSel* and *endSel*, the contents within the current text selection is evaluated.

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

**Note:** If *startSel* is greater than *endSel* (except when *endSel* is 0), the command does nothing and the *OK* variable is set to 0.

The optional *startBlock* and *endBlock* parameters retrieve the position of the first and last character of the first homogenous block identified in the object or the selection of the object. For example, if the selection contains an expression and then plain text, *startBlock* and *endBlock* will return the limits of the expression. You can make a loop to process all the blocks of the selection.

This command returns a value designating the type of contents identified. You can compare this value with the following constants, found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST Expression type | Longint | 2 | Selection contains only an expression reference |
| ST Mixed type | Longint | 3 | Selection contains at least two different types of contents |
| ST Picture type | Longint | 6 | Selection contains only a picture (4D Write Pro areas only) |
| ST Plain type | Longint | 0 | Selection contains text and no references |
| ST Unknown tag type | Longint | 4 | Selection contains only an unknown tag type |
| ST URL type | Longint | 1 | Selection contains only a URL reference |
| ST User type | Longint | 5 | Selection contains only a custom reference |

## Example

You want to display context-menu commands based on the type of contents selected in the area.

```
Case of
   :(Form event=On Clicked)
 //we retrieve the selection
      GET HIGHLIGHT(*;"myText";startSel;endSel)
      If(Contextual click & (Macintosh control down=False)) //calls the context menu
         If(startSel=endSel) // no contents selected
 //we enable only certain commands
            DISABLE MENU ITEM(<>menu_STYLEDTEXT;2)
            DISABLE MENU ITEM(<>menu_STYLEDTEXT;4)
            ENABLE MENU ITEM(<>menu_STYLEDTEXT;6)
            ...
         Else // we get the content type
            CT_Texttype:=ST Get content type(*;"myText";startSel;endSel)
            Case of // processing of different types
               :(CT_Texttype=ST URL type)
                  DISABLE MENU ITEM(<>menu_STYLEDTEXT;6)
                  ENABLE MENU ITEM(<>menu_STYLEDTEXT;7)
                  ...
               :(CT_Texttype=ST Expression type)
                  DISABLE MENU ITEM(<>menu_STYLEDTEXT;6)
                  DISABLE MENU ITEM(<>menu_STYLEDTEXT;7)
                  ...
               Else
                  ENABLE MENU ITEM(<>menu_STYLEDTEXT;6)
                  DISABLE MENU ITEM(<>menu_STYLEDTEXT;7)
                  ...
            End case
         End if
         GET MOUSE($xCoord;$yCoord;$ButtonState)
         $AlphaVar:=Dynamic pop up menu(<>menu_STYLEDTEXT;"";$xCoord;$yCoord)
         startSel:=-3
         endSel:=-3
      End if
      ...
   End if
```

# ST Get expression

ST Get expression ( {* ;} object {; startSel {; endSel}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |
| Function result | Text | ⇐ | Expression label |

## Description

The **ST Get expression** command returns the first expression found in the current selection of the styled text field or variable designated by the *object* parameter.

The command returns the label of the expression as it was inserted into the object (for example "mymethod" or "[table1]field1"). The current value of the expression is not returned.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). During execution, if the object has the focus, the command returns the information of the object being edited; if the object does not have the focus, the command returns the information of the object's data source (variable or field).
If you omit the * parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string. During execution, the command returns the information of the variable or field.

The optional *startSel* and *endSel* parameters designate a selection of text in *object*. The *startSel* and *endSel* values express a plain text selection, without taking into account any style tags that may be present.

- If you pass *startSel* and *endSel*, **ST Get expression** looks for the expression within this selection.
- If you only pass *startSel* or if the value of *endSel* is greater than the total number of characters in *object*, the command looks for the expression between *startSel* and the end of the text.
- If you omit *startSel* and *endSel*, the command looks for the expression within the current text selection.

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

**Note:** If *startSel* is greater than *endSel* (except when *endSel* is 0), the command does nothing and the *OK* variable is set to 0.

If there is no expression found in the selection, the command returns an empty string.

## Example 1

When there is a double-click event, you check that there is in fact an expression, and if so, you display a dialog where you have retrieved its values so that the user can modify them:

```
Case of
   :(Form event=On Double Clicked)
      GET HIGHLIGHT(*;"StyledText_t";startSel;endSel)
      If(ST Get content type(*;"StyledText_t";startSel;endSel)=ST Expression type)
         vExpression:=ST Get expression(*;"StyledText_t";startSel;endSel)
         $winRef:=Open form window("Dial_InsertExpr";Movable form dialog box;Horizontally centered;Vertically centered;*)
         DIALOG("Dial_InsertExpr")
         If(OK=1)
```

```
            ST INSERT EXPRESSION(*;"StyledText_t";vExpression;startSel;endSel)
            HIGHLIGHT TEXT(*;"StyledText_t";startSel;endSel)
         End if
      End if
End case
```

## Example 2

You want to execute a 4D method when a user link is clicked:

```
Case of
   :(Form event=On Clicked)
 //we retrieve the selection
      HIGHLIGHT TEXT(*;"myText";startSel;endSel)
      If(startSel#endSel) //there is selected content
 //we get the content type
         $CT_type:=ST Get content type(*;"myText";startSel;endSel)
         If($CT_type=ST User type) //this is a user link
            MyMethod //we execute a 4D method
         End if
      End if
End case
```

# ST GET OPTIONS

ST GET OPTIONS ( {* ;} object ; option ; value {; option2 ; value2 ; ... ; optionN ; valueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| option | Longint | ⇒ | Option to get |
| value | Longint | ⇐ | Current value of option |

## Description

The **ST GET OPTIONS** command gets the current value of one or more operating options for the styled text field or variable designated by the *object* parameter.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). During execution, if the object has the focus, the command returns the information of the object being edited; if the object does not have the focus, the command returns the information of the object's data source (variable or field).
If you omit the * parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string. During execution, the command returns the information of the variable or field.

Pass the code of the option to get in the *option* parameter. The command returns the current value of this option in *value*.
For both these parameters, you can use the following constants, found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST Expressions display mode | Longint | 1 | The *value* parameter can contain ST Values or ST References |
| ST References | Longint | 1 | Display source strings of expressions |
| ST Values | Longint | 0 | Display computed values of expressions |

# ST Get plain text

ST Get plain text ( {* ;} object {; refMode} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| refMode | Longint | ⇒ | Mode for handling references found in the text |
| Function result | Text | ⇐ | Text without tags |

## Description

The **ST Get plain text** command removes any style tags from the text variable or field designated by the * and *object* parameters and returns the plain text.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (string). During execution, if the object has the focus, the command returns information about the object being edited; however, when the object does not have the focus, the command returns information about the data source (field or variable) of the object.
If you omit the * parameter, this indicates that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference instead of a string and during execution, the command returns information about this field or variable.

The optional *refMode* parameter indicates the way that references found in *object* must be returned. In *refMode*, pass one of the following constants, found in the "**Multistyle Text**" theme (you can pass a single constant or a combination):

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST 4D Expressions as sources | Longint | 2 | The original string of 4D expression references is returned |
| ST 4D Expressions as values | Longint | 1 | 4D expression references are returned in their evaluated form (default functioning in forms) |
| ST References as spaces | Longint | 0 | Each reference is returned as a non-breaking space character (default operation, used by other commands) |
| ST Tags as plain text | Longint | 64 | The label of the tag is returned in plain text. For example for the tag '<img src="test.jpg" alt="picture">my picture</img>', the plain text is "my picture" (default functioning in forms) |
| ST Tags as XML code | Longint | 128 | The XML code of the tag is returned in plain text. For example for the tag '<img src="test.jpg" alt="picture">my picture</img>', the plain text is '<img src="test.jpg" alt="picture">my picture</img>' |
| ST Text displayed with 4D Expression sources | Longint | 86 | Returns the text as it is shown in the forms with the original string of the 4D expressions.<br>Corresponds a predefined combination of constants 2+4+16+64. |
| ST Text displayed with 4D Expression values | Longint | 85 | Returns the text as it is shown in the forms with the 4D expressions in their evaluated form.<br>Corresponds to a predefined combination of constants 1+4+16+64. |
| ST URL as labels | Longint | 4 | The visible label of URLs is returned, for example "Visit our Web site" (default functioning in forms) |
| ST URL as links | Longint | 8 | The link is returned, for example "http://www.4d.com" |
| ST User links as labels | Longint | 16 | The visible label of the user link is returned (default functioning in forms) |
| ST User links as links | Longint | 32 | The contents of the user link is returned |

**Note:** Since plain text remains the same regardless of the values passed in the *refMode* parameter, the optional *refMode* parameter is only useful when the text contains references.

## Example 1

You are looking for the text "very nice" among the values of a multistyle text field. The value was stored in the following form: "The weather is very nice **today**".

```
QUERY BY FORMULA([Comments];ST Get plain text([Comments]Weather)="@very nice@")
```

**Note:** In this context, the following statement will not give the desired result because the text is saved with style tags:

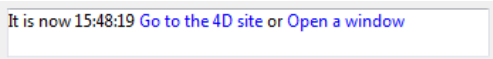
```
QUERY([Comments];[Comments]Weather="@very nice@")
```

## Example 2

Given the following text placed in the multi-style area entitled "MyArea":

```
<span>It is now <span style="-d4-ref:'Current time:C178'"> </span> <a href="http://www.4d.com">Go to the 4D site</a> or
<span style="-d4-ref-user:'openW'">Open a window</span></span>
```

This text is displayed:




If you execute the following code:

```
$txt :=ST Get plain text(*;"myArea";ST References as spaces)
 // $txt = "It is now    or "  (spaces)
$txt :=ST Get plain text(*;"myArea";ST 4D Expressions as values)
 // $txt = "It is now 15:48:19  or  "
$txt :=ST Get plain text(*;"myArea";ST 4D Expressions as sources)
 // $txt = "It is now Current time  or  "
$txt :=ST Get plain text(*;"myArea";ST URL as links)
 //$txt = "It is now   http://www.4d.com or  "
$txt :=ST Get plain text(*;"myArea";ST Text displayed with 4D Expression values)
 //$txt = "It is now 15:48:19 Go to the 4D site or Open a window"
$txt :=ST Get plain text(*;"myArea";ST Text displayed with 4D Expression sources)
 //$txt = "It is now Current time Go to 4D site or Open a window"
$txt :=ST Get plain text(*;"myArea";ST User links as labels)
 //$txt = "It is now    or Open a window"
$txt :=ST Get plain text(*;"myArea";ST User links as links)
 //$txt = "It is now    or openW"
```

## System variables and sets

After this command is executed, the OK variable is set to 1 if no error occurred; otherwise, it is set to 0. This is the case more particularly when style tags are not evaluated properly (incorrect or missing tags).

In the case of an error, the variable is not changed. When an error occurs on a variable when text is being evaluated, 4D transforms the text into plain text; as a result, the <, > and & characters are converted into HTML entities.

# ST Get text

ST Get text ( {* ;} object {; startSel {; endSel}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Text field or variable (if * is omitted) |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |
| Function result | Text | ⇐ | Text including style tags |

## Description

The **ST Get text** command returns the styled text found in the text field or variable designated by the *object* parameter.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (string). During execution, if the object has the focus, the command returns information about the object being edited; however, when the object does not have the focus, the command returns information about the data source (field or variable) of the object.
If you omit the * parameter, this indicates that the *object* parameter is a field or a variable. In this case, you pass a field or variable reference instead of a string and during execution, the command returns information about this field or variable.

The command returns the text with any style tags that are associated with it, which means, for example, that you can copy and paste text while keeping its style.

The optional *startSel* and *endSel* parameters let you designate a selection of text in object. The *startSel* and *endSel* values give a selection of plain text, without taking any style tags found in the text into account.

- If you omit *startSel* and *endSel*, **ST Get text** returns all the text contained in *object*,
- If you pass *startSel* and *endSel*, **ST Get text** returns the selection of text set by these limits.

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

If the values of *startSel* and *endSel* are equal or if *startSel* is greater than *endSel*, an error is returned.

## System variables and sets

After this command is executed, the OK variable is set to 1 if no error occurred; otherwise, it is set to 0. This is the case more particularly when style tags are not evaluated properly (incorrect or missing tags).

In the case of an error, the variable is not changed. When an error occurs on a variable when text is being evaluated, 4D transforms the text into plain text; as a result, the <, > and & characters are converted into HTML entities.

# ST GET URL

ST GET URL ( {* ;} object ; urlText ; urlAddress {; startSel {; endSel}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| urlText | Text | ⇐ | Visible text of URL |
| urlAddress | Text | ⇐ | URL address |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |

## Description

The **ST GET URL** command returns the text label and address of the first URL detected in the styled text field or variable designated by the *object* parameter.

The text label and address are returned in the *urlText* and *urlAddress* parameters. If the selection does not contain a URL, empty strings are returned in these parameters.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). During execution, if the object has the focus, the command returns the information of the object being edited; if the object does not have the focus, the command returns the information of the object's data source (variable or field).
If you omit the * parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string. During execution, the command returns the information of the variable or field.

The optional *startSel* and *endSel* parameters designate a selection of text in object. The *startSel* and *endSel* values express a plain text selection, without taking into account any style tags that may be present.

- If you pass *startSel* and *endSel*, **ST GET URL** looks for the URL within this selection.
- If you only pass *startSel* or if the value of *endSel* is greater than the total number of characters in *object*, the command looks for the URL between *startSel* and the end of the text.
- If you omit *startSel* and *endSel*, the command looks for the URL within the current text selection.

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

**Note:** If *startSel* is greater than *endSel* (except when *endSel* is 0), the command does nothing and the *OK* variable is set to 0.

## Example

When there is a double-click event, you check that there is in fact an URL, and if so, you display a dialog where you have retrieved its values so that the user can modify them:

```
Case of
   :(Form event=On Double Clicked)
      GET HIGHLIGHT(*;"StyledText_t";startSel;endSel)
      If(ST Get content type(*;"StyledText_t";startSel;endSel)=ST URL type) //URL
         ST GET URL(*;"StyledText_t";vTitle;vURL;startSel;endSel)
         $winRef:=Open form window("Dial_InsertURL";Movable form dialog box;Horizontally centered;Vertically centered;*)
         SET WINDOW TITLE("URL settings")
         DIALOG("Dial_InsertURL")
         If(OK=1)
```

```
            ST INSERT URL(*;"StyledText_t";vTitle;vURL;startSel;endSel)
            HIGHLIGHT TEXT(*;"StyledText_t";startSel;startSel+1)
         End if
      End if
End case
```

# ST INSERT EXPRESSION

ST INSERT EXPRESSION ( {* ;} object ; expression {; startSel {; endSel}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| expression | Text | ⇒ | Expression and (optional) format to insert |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |

## Description

The **ST INSERT EXPRESSION** command inserts a reference to the *expression* in the styled text field or variable designated by the *object* parameter.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In the *expression* parameter, you pass the 4D expression to evaluate in the *object*. A valid 4D expression is a string returning a value. *expression* can be a field, a variable, a 4D command, a statement returning a value, a project method, and so on.

The expression must be placed in quotation marks ("").

**Note:** The *expression* parameter cannot be a Picture type variable.

If *expression* returns a value containing carriage returns and tabs, 4D formats the text according to the object hosting the expression; carriage return characters are interpreted as line breaks.

You can format the expression by including formatting information in the *expression* parameter. In this case, the parameter must be in the form:

```
"String(value:format)"
```

... where *value* contains the expression itself and *format* contains the format to apply. The *format* parameter can have the following values:

- for numbers: any number display format (existing or not), for example "###,##".
- for dates: a number designating an existing date format. You can use the constants of the "**Date Display Formats**" theme, for example System date short .
- for times: a number designating an existing time format. You can use the constants of the "**Time Display Formats**" theme, for example System time short .

For example:

```
"String([Table_1]Field_1;System date short)"
```

By default, the expression **values** are displayed in the multi-style text areas. You can force the display of the **references** instead using the **ST SET OPTIONS** command.

The optional *startSel* and *endSel* parameters designate a selection of text in *object*. The *startSel* and *endSel* values express a plain text selection, without taking into account any style tags that may be present.

- If you only pass *startSel*, the result of the expression is inserted at the specified location.
- If you omit *startSel* and *endSel*, the result of the expression is inserted at the location of the cursor.
- If you pass *startSel* and *endSel*, **ST INSERT EXPRESSION** replaces the text in this selection with the result of the *expression*. If the value of *endSel* is greater than the total number of characters in the object, all the characters between *startSel* and the end of the text are replaced by the result of the *expression*.

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

**Note:** If *startSel* is greater than *endSel* (except when *endSel* is 0), the command does nothing and the *OK* variable is set to 0.

## Example

You want to replace the highlighted text with the result of a project method:

```
ST INSERT EXPRESSION(*;"myText";"MyMethod";ST Start highlight;ST End highlight)
```

# ST INSERT URL

ST INSERT URL ( {* ;} object ; urlText ; urlAddress {; startSel {; endSel}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| urlText | Text | ⇒ | Visible text of URL |
| urlAddress | Text | ⇒ | URL address |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |

## Description

The **ST INSERT URL** command inserts a URL link in the styled text field or variable designated by the *object* parameter.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

In the *urlText* parameter, pass the visible text of the URL, as it should appear in the object. For example, text labels such as "4D Web Site" or "Follow this link for more information" can be used. You can also use the address itself, for instance "http://www.4d.com".

In the *urlAddress* parameter, pass the complete address you want the browser page to connect to, for example "http://www.4D.com".

The optional *startSel* and *endSel* parameters designate a selection of text in *object*. The *startSel* and *endSel* values express a plain text selection, without taking into account any style tags that may be present.

- If you only pass *startSel*, *urlText* is inserted at the specified location.
- If you omit *startSel* and *endSel*, *urlText* is inserted at the location of the cursor.
- If you pass *startSel* and *endSel*, **ST INSERT URL** replaces the text in this selection with the *urlText*. If the value of *endSel* is greater than the total number of characters in the object, all the characters between *startSel* and the end of the text are replaced with the *urlText*.

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

**Note:** If *startSel* is greater than *endSel* (except when *endSel* is 0), the command does nothing and the *OK* variable is set to 0.

Once the link is inserted, it is active: using **Ctrl+click** (Windows) or **Command+click** (OS X) on its label opens a page of the default browser at the address specified in the *urlAddress* parameter.

## Example

You want to insert a link to the 4D Web site to replace the text selected in the object:

```
vTitle:="4D Web Site"
vURL:="http://www.4d.com/"
ST INSERT URL(*;"myText";vTitle;vURL;ST Start highlight;ST End highlight)
```

# ST SET ATTRIBUTES

ST SET ATTRIBUTES ( {* ;} object ; startSel ; endSel ; attribName ; attribValue {; attribName2 ; attribValue2 ; ... ; attribNameN ; attribValueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Text field or variable (if * is omitted) |
| startSel | Longint | ⇒ | Start of new text selection |
| endSel | Longint | ⇒ | End of new text selection |
| attribName | String | ⇒ | Attribute to set |
| attribValue | String, Longint | ⇒ | New value of attribute |

## Description

The **ST SET ATTRIBUTES** command can be used to modify one or more style attributes in the form object(s) designated by *object*.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (string). During execution, when the object has the focus, the command only applies to the object being edited and not to its data source (field or variable). The changes are only transferred to the source (and to any other objects using this same source) when the object being edited is validated either by losing the focus or with the **Enter** key. When the object does not have the focus, the command applies directly to the data source and the changes are immediately transferred to any other objects using the same source.
If you omit the * parameter, this indicates that the *object* parameter is a field or variable and you pass a field or variable reference instead of a string. The command applies directly to the field or variable and changes are immediately transferred to all the objects using this source, including the object with the focus.

**Note:** You can only use style attributes with Text type fields. Since Alpha type fields have a preset length, adding style tags would lead to a loss of data.

The definition of an attribute is carried out via the insertion or modification of HTML style tags within the text (for more information about this point, refer to the *Design Reference* manual). Note that **ST SET ATTRIBUTES** inserts style tags in all cases, even if the *object* designates text objects of the form that do not have the Multistyle property.

The *startSel* and *endSel* parameters can be used to designate the selection of text to which to apply the style modification(s) within the *object*. In *startSel*, you pass the position of the first character to be modified and in *endSel*, you pass the position of the last character to be modified plus one (the last character passed is not included in the modification). You can pass 0 in *endSel* to designate automatically the last character of the text (pass 1 in *startSel* to designate the first character of the text).
If the value of *endSel* is greater than the number of characters in the object, all the characters between *startSel* and the end of the text are modified. If *startSel* is greater than *endSel* (except when *endSel* value is 0, see above), the command does nothing and the OK variable is set to 0.
The *startSel* and *endSel* values do not take any style tags already present in the area into account. They are evaluated on the basis of raw text (text where style tags have been filtered).

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

In *attribName* and *attribValue*, pass the name and the value, respectively, of the attribute to be modified. You can pass as many attribute/value pairs as you want. To specify the *attribName* parameter, use the predefined constants placed in the

**Multistyle Text Attributes** theme. The value passed in the *attribValue* parameter depends on the *attribName* parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Attribute background color | Longint | 8 | *attribValue*=Hexadecimal values or HTML color names (Windows only) |
| Attribute bold style | Longint | 1 | *attribValue*=0: remove bold attribute from selection<br>*attribValue*=1: apply bold attribute to selection |
| Attribute font name | Longint | 5 | *attribValue*=Font family name (string) |
| Attribute italic style | Longint | 2 | *attribValue*=0: remove italic attribute from selection<br>*attribValue*=1: apply italic attribute to selection |
| Attribute strikethrough style | Longint | 3 | *attribValue*=0: remove strikethrough attribute from selection<br>*attribValue*=1: apply strikethrough attribute to selection |
| Attribute text color | Longint | 7 | *attribValue*=Hexadecimal values or HTML color names |
| Attribute text size | Longint | 6 | *attribValue*=Number of points (number) |
| Attribute underline style | Longint | 4 | *attribValue*=0: remove underline attribute from selection<br>*attribValue*=1: apply underline attribute to selection |

**Colors**

If you pass the Attribute text color or Attribute background color constants in *attribName*, you must pass a string containing either an HTML color name or a hexadecimal color value in *attribValue*:

| HTML color name | Hexa value |
|---|---|
| Aqua | #00FFFF |
| Black | #000000 |
| Blue | #0000FF |
| Fuchsia | #FF00FF |
| Gray | #808080 |
| Green | #008000 |
| Lime | #00FF00 |
| Maroon | #800000 |
| Navy | #000080 |
| Olive | #808000 |
| Purple | #800080 |
| Red | #FF0000 |
| Silver | #C0C0C0 |
| Teal | #008080 |
| White | #FFFFFF |
| Yellow | #FFFF00 |

## Example

In this example, we modify the size and color of the text as well as the bold and underline attributes of the characters 2 to 4 of the field:

```
ST SET ATTRIBUTES([MyTable]MyField;2;5;Attribute font name;"Arial";Attribute text size;10;Attribute underline style;1;Attribute bold style;1;Attribute text color;"Blue")
```

## System variables and sets

After this command is executed, the OK variable is set to 1 if no error occurred; otherwise, it is set to 0. This is the case more particularly when style tags are not evaluated properly (incorrect or missing tags).

In the case of an error, the variable is not changed. When an error occurs on a variable when text is being evaluated, 4D transforms the text into plain text; as a result, the <, > and & characters are converted into HTML entities.

# ST SET OPTIONS

ST SET OPTIONS ( {* ;} object ; option ; value {; option2 ; value2 ; ... ; optionN ; valueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a field or variable |
| object | Form object | ⇒ | Object name (if * is specified) or Field or variable (if * is omitted) |
| option | Longint | ⇒ | Option to set |
| value | Longint | ⇒ | New value of option |

## Description

The **ST SET OPTIONS** command modifies one or more operating options for the styled text field or variable designated by the *object* parameter.

Passing the optional * parameter indicates that the *object* parameter is an object name (string). If you do not pass this parameter, it indicates that the *object* parameter is a field or variable. In this case, you pass a field or variable reference instead of a string (field or variable object only).

Pass the code of the option to modify in *option* and its new value in *value*.

The *option* parameter supports the following constant found in the "**Multistyle Text**" theme:

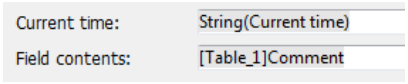
| Constant | Type | Value | Comment |
|---|---|---|---|
| ST Expressions display mode | Longint | 1 | The *value* parameter can contain ST Values or ST References |

In the *value* parameter, you can pass one of the following constants:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST References | Longint | 1 | Display source strings of expressions |
| ST Values | Longint | 0 | Display computed values of expressions |

Display of values:


Display of expressions:




## Example

The following code lets you switch the display mode of the area:

```
ST GET OPTIONS(*;"StyledText_t";ST Expressions display mode;$exprValue)
If($exprValue=1)
   ST SET OPTIONS(*;"StyledText_t";ST Expressions display mode;ST Values)
Else
   ST SET OPTIONS(*;"StyledText_t";ST Expressions display mode;ST References)
End if
```

# ST SET PLAIN TEXT

ST SET PLAIN TEXT ( {* ;} object ; newText {; startSel {; endSel}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Variable or field (if * is omitted) |
| newText | Text | ⇒ | Text to insert |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |

## Description

The **ST SET PLAIN TEXT** command inserts the text passed in the *newText* parameter into the styled text field or variable designated by the *object* parameter. This command only applies to the plain text of the*object* parameter, without modifying any style tags that it contains.

Unlike the **ST SET TEXT** command, **ST SET PLAIN TEXT** only inserts plain text. You must not pass text with style tags  in *newText* must not have any style tags. If it contains the <, > or & characters, they are considered as standard characters and converted into HTML entities:

- '&' is converted to &
- '<' is converted to <
- '>' is converted to >

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (string). During execution, when the object has the focus, the command only applies to the object being edited and not to its data source (field or variable). The changes are only transferred to the source (and to any other objects using this same source) when the object being edited is validated either by losing the focus or with the **Enter** key. When the object does not have the focus, the command applies directly to the data source and the changes are immediately transferred to any other objects using the same source.
If you omit the * parameter, this indicates that the *object* parameter is a field or variable and you pass a field or variable reference instead of a string. The command applies directly to the field or variable and changes are immediately transferred to all the objects using this source, including the object with the focus.

In *newText*, pass the plain text to be inserted.

The optional *startSel* and *endSel* parameters let you designate a selection of text in *object*. The *startSel* and *endSel* values give a selection of plain text, without taking any style tags found in the text into account. The action of the command varies according to the optional *startSel* and *endSel* parameters:

- If you omit *startSel* and *endSel*, **ST SET PLAIN TEXT** replaces all the text of the *object* by *newText*,
- If you only pass *startSel* or if the values of *startSel* and *endSel* are equal, **ST SET PLAIN TEXT** inserts the *newText* text into *object* beginning at *startSel*,
- If you pass both *startSel* and *endSel*, **ST SET PLAIN TEXT** replaces the plain text set by these limits with the *newText* text.
- You can pass 0 in *endSel* to designate automatically the last character of the text (pass 1 in *startSel* to designate the first character of the text).

4D provides predefined constants that you can use to automatically designate the limits of the selection in the *startSel* and *endSel* parameters. These constants are available in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a field or variable reference, the command is applied to all the text of the object.

The style of the first character replaced will be used for all of the *newText* text.

If *startSel* is greater than *endSel* (except when *endSel* value is 0, see above), the text is not modified and the OK variable is set to 0.

## Example

Given the following rich text (multi-style) variable:

Please remember that the **X company has made a commitment** to you.

You want to insert company names that are stored in a text field. These names can contain, for example, the "&" character. In this case, you will need to use the **ST SET PLAIN TEXT** command:

```
ST SET PLAIN TEXT(myStyledText;[Company]Name;33;34)
```

Here is the result:

Please remember that the **Smith & Jones company has made a commitment** to you.

Here is the plain text contained in the variable:

```
Please remember that the <SPAN STYLE="font-weight:bold"> Smith &amp;amp; Jones </SPAN> <SPAN STYLE="font-weight:bold">company has made a commitment</SPAN> to you.
```

You can see that the inserted text was enclosed within an additional pair of style tags. These tags correspond to the style of the characters before they were inserted. This mechanism is a way of guaranteeing the correct display of rich text fields in all cases.

**Note:** If you had used the **ST SET TEXT** command in this case, 4D would not have inserted anything because the presence of the non-encoded "&" character would prevent the interpretation of the style tags found in the variable. For more information, refer to the description of this command.

## System variables and sets

After this command is executed, the OK variable is set to 1 if no error occurred; otherwise, it is set to 0. This is the case more particularly when style tags are not evaluated properly (incorrect or missing tags).

In the case of an error, the variable is not changed. When an error occurs on a variable when text is being evaluated, 4D transforms the text into plain text; as a result, the <, > and & characters are converted into HTML entities.

# ST SET TEXT

ST SET TEXT ( {* ;} object ; newText {; startSel {; endSel}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable or a field |
| object | Form object | ⇒ | Object name (if * is specified) or Text field or variable (if * is omitted) |
| newText | Text | ⇒ | Text to insert |
| startSel | Longint | ⇒ | Start of selection |
| endSel | Longint | ⇒ | End of selection |

## Description

The **ST SET TEXT** command inserts the text passed in the *newText* parameter into the styled text field or variable designated by the *object* parameter. This command only applies to the plain text of the object parameter, without modifying any style tags that it contains. It can be used to modify, by programming, styled text displayed on screen.

If you pass the optional * parameter, this indicates that the *object* parameter is an object name (string). During execution, when the object has the focus, the command only applies to the object being edited and not to its data source (field or variable). The changes are only transferred to the source (and to any other objects using this same source) when the object being edited is validated either by losing the focus or with the **Enter** key. When the object does not have the focus, the command applies directly to the data source and the changes are immediately transferred to any other objects using the same source.
If you omit the * parameter, this indicates that the *object* parameter is a field or variable and you pass a field or variable reference instead of a string. The command applies directly to the field or variable and changes are immediately transferred to all the objects using this source, including the object with the focus.

In *newText*, pass the text to be inserted. The **ST SET TEXT** command is intended for working with rich text (multistyle) containing  type tags. In all other cases (more particularly, when working with plain text that contains the <, > or & characters), you must use the **ST SET PLAIN TEXT** command. If you pass plain text containing the <, > or & characters to the **ST SET TEXT** command, it does nothing. This is necessary behavior because if you insert a string such as "a<b" directly into rich text, it will distort the internal analysis of the  tags. In this case, "<" characters must be encoded beforehand as "<", which can be done using the **ST SET PLAIN TEXT** command (see also the example of this command).

The optional *startSel* and *endSel* parameters let you designate a selection of text in *object*. The *startSel* and *endSel* values give a selection of plain text, without taking any style tags found in the text into account. The action of the command varies according to the optional *startSel* and *endSel* parameters:

- If you omit *startSel* and *endSel*, **ST SET TEXT** replaces all the text of the *object* by *newText*,
- If you only pass *startSel* or if the values of *startSel* and *endSel* are equal, **ST SET TEXT** inserts the *newText* text into object beginning at *startSel*,
- If you pass both *startSel* and *endSel*, **ST SET TEXT** replaces the plain text set by these limits with the *newText* text.
- You can pass 0 in *endSel* to designate automatically the last character of the text (pass 1 in *startSel* to designate the first character of the text).

4D provides predefined constants so that you can designate the selection limits automatically in the *startSel* and *endSel* parameters. These constants are found in the "**Multistyle Text**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |

(*) You must pass an object name in *object* to be able to use this constant. If you pass a reference to a field or variable, the command is applied to all the text of the object.

**Note:** If *startSel* is greater than *endSel* (except if *endSel* value is 0, see above), the text is not modified and the OK variable is set to 0.

## System variables and sets

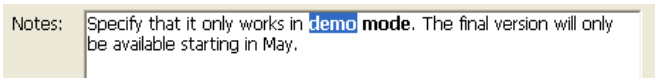
After this command is executed, the OK variable is set to 1 if no error occurred; otherwise, it is set to 0. This is the case more particularly when style tags are not evaluated properly (incorrect or missing tags).

In the case of an error, the variable is not changed. When an error occurs on a variable when text is being evaluated, 4D transforms the text into plain text; as a result, the <, > and & characters are converted into HTML entities.

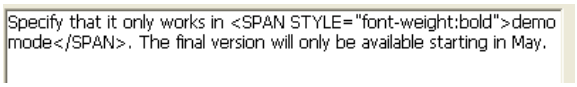
## Example 1

You want to replace the styled text selected by the user with the contents of a variable.

Here is the selected text:



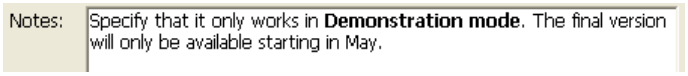

The following contents are stored in the field:

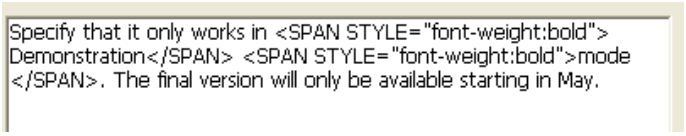



After executing this code:

```
vtempo:="Demonstration"
GET HIGHLIGHT([Products]Notes;vStart;vEnd)
ST SET TEXT([Products]Notes;vtemp;vStart;vEnd)
```

The field and its contents are as follows:




## Example 2

Refer to the example of the **ST SET PLAIN TEXT** command.

# Subrecords

- Get subrecord key
- *_o_ALL SUBRECORDS*
- *_o_APPLY TO SUBSELECTION*
- *_o_Before subselection*
- *_o_CREATE SUBRECORD*
- *_o_DELETE SUBRECORD*
- *_o_End subselection*
- *_o_FIRST SUBRECORD*
- *_o_LAST SUBRECORD*
- *_o_NEXT SUBRECORD*
- *_o_ORDER SUBRECORDS BY*
- *_o_PREVIOUS SUBRECORD*
- *_o_QUERY SUBRECORDS*
- *_o_Records in subselection*

# Get subrecord key

Get subrecord key ( idField ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| idField | Field | ➡ | "Subtable Relation" or "Longint" type field of a former subtable relation |
| Function result | Longint | ⮌ | Internal key of relation |

## Description

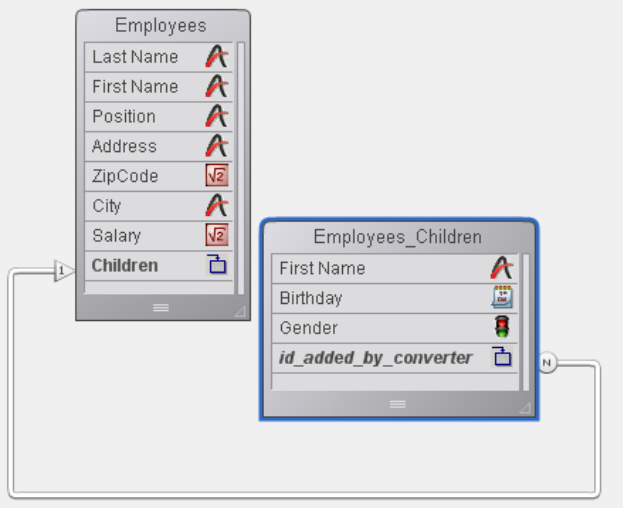
The **Get subrecord key** command facilitates the migration of 4D code using converted subtables to standard code for working with tables.

**Reminder:** Beginning with version 11 of 4D, subtables are not supported. When a older database is converted, any existing subtables are transformed into standard tables that are linked with the original tables by an automatic relation. The former subtable becomes the Many table and the original table is the One table. In the One table, the former subtable field is transformed into a special field of the "Subtable Relation" type and in the Many field, a special "Subtable Relation" type field is added named “id_added_by_converter”.

This allows converted databases to function but we strongly recommend that you replace any subtable mechanisms in your converted databases by those used for standard tables.

The first step in this process consists in deleting the special automatic relations, which permanently disables the mechanisms inherited from subtables. After this you need to rewrite the associated code. The **Get subrecord key** command accompanies this rewriting by returning the internal ID used by the relation. This internal ID makes the actual relation unnecessary and you can then work with the selection of the former subtable even when the relation is no longer present.

Let's look for example at the following converted structure:




In 4D, the following code still works but it must be updated:

```
ALL SUBRECORDS([Employees]Children)
$total:=Records in subselection([Employees]Children)
vFirstnames:=""
For($i;1;$total)
   vFirstnames:=vFirstnames+[Employees]Children'FirstName+" "
   NEXT SUBRECORD([Employees]Children)
End for
```

You can now replace this code with:

```
QUERY([Employees_Children];[Employees_Children]id_added_by_converter=Get subrecord key([Employees]Children))
$total:=Records in selection([Employees_Children])
vFirstnames:=""
For($i;1;$total)
   vFirstnames:=vFirstnames+[Employees_Children]FirstName+" "
   NEXT RECORD(Employees_Children)
End for
```

**Note: Get subrecord key** returns 0 if there is no current recorded loaded when it is executed.

The second piece of code has the advantage of using standard 4D commands and it works the same way whether the relation is present or not. When you remove the relation, the command simply returns the key value stored in the Longint field.

In the *idField* parameter, the command accepts either a field of the Subtable Relation type (if the relation still exists) or of the Longint type (if the relation has been removed). In any other case, an error is generated.

This lets you write the transition code. During the final stage of upgrading the application, you can remove the calls to this command.

## Assigning values to the id_added_by_converter field

Starting with 4D v14 R3, you can assign a value to the "id_added_by_converter" field. Previously, this value could only be assigned by 4D itself, which forced developers to use obsolete commands such as **_o_CREATE SUBRECORD** to be able to add records into converted subtables.

Thanks to this possibility, you can convert former databases containing subtables progressively: at first, you can keep the special "Subtable relation", while adding or modifying related records as if they were standard ones. Then, once all your methods are up to date, you can replace this special relation with a standard one, without having to change your code.

For example, with the structure above you can write:

```
CREATE RECORD([Employees])
[Employees]LastName:="Jones"
CREATE RECORD([Employees_Children])
[Employees_Children]FirstName:="Natacha"
[Employees_Children]BirthDate:=!12/24/2013!
[Employees_Children]id_added_by_converter:=Get subrecord key([Employees]Children)
SAVE RECORD([Employees_Children])
SAVE RECORD([Employees]
```

This code will work with either a special relation or a standard one.

# _o_ALL SUBRECORDS

_o_ALL SUBRECORDS ( subtable )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable in which to select all subrecords |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_APPLY TO SUBSELECTION

_o_APPLY TO SUBSELECTION ( subtable ; statement )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable to which to apply the formula |
| statement | Statement | ⇒ | One line of code or a method |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_Before subselection

_o_Before subselection ( subtable ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable for which to test if subrecord pointer is before the first selected subrecord |
| Function result | Boolean | ⇐ | Yes (TRUE) or No (FALSE) |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_CREATE SUBRECORD

_o_CREATE SUBRECORD ( subtable )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable for which to create a new subrecord |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_DELETE SUBRECORD

_o_DELETE SUBRECORD ( subtable )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable from which to delete the current subrecord |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_End subselection

_o_End subselection ( subtable ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable for which to test if subrecord pointer is after the last selected subrecord |
| Function result | Boolean | ⇐ | Yes (TRUE) or No (FALSE) |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_FIRST SUBRECORD

_o_FIRST SUBRECORD ( subtable )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇨ | Subtable in which to move to the first selected subrecord |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_LAST SUBRECORD

_o_LAST SUBRECORD ( subtable )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable in which to move to the last selected subrecord |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_NEXT SUBRECORD

_o_NEXT SUBRECORD ( subtable )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇨ | Subtable in which to move to the next selected subrecord |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

## _o_ORDER SUBRECORDS BY

_o_ORDER SUBRECORDS BY ( subtable ; subfield {; > or <}{; subfield2 ; > or <2 ; ... ; subfieldN ; > or <N} )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable by which to order the selected subrecords |
| subfield | Subfield | ⇒ | Subfield on which to order by for each level |
| > or < | Operator | ⇒ | Ordering direction for each level: > to order in ascending order or < to order in descending order |

### Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_PREVIOUS SUBRECORD

_o_PREVIOUS SUBRECORD ( subtable )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable in which to move to the previous selected subrecord |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_QUERY SUBRECORDS

_o_QUERY SUBRECORDS ( subtable ; queryFormula )

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable to search |
| queryFormula | Boolean | ⇒ | Query formula |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# _o_Records in subselection

_o_Records in subselection ( subtable ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| subtable | Subtable | ⇒ | Subtable for which to count number of subrecords |
| Function result | Longint | ⇒ | Number of subrecords in current subselection |

## Compatibility note

Subtables are no longer supported starting with version 11 of 4D. A compatibility mechanism ensures the functioning of this command in converted databases; however, it is strongly recommended to replace any subtables with standard related tables.

# SVG

- Overview of SVG commands
- SVG EXPORT TO PICTURE
- SVG Find element ID by coordinates
- SVG Find element IDs by rect
- SVG GET ATTRIBUTE
- SVG SET ATTRIBUTE
- SVG SHOW ELEMENT

## Overview of SVG commands

---

SVG (Scalable Vector Graphics) is an XML-based file format (.svg extension) that is used to describe vector graphics. SVG is most commonly used for publishing statistical or cartographic data.

These files can be viewed in Web browsers either natively or via plug-ins. 4D includes an SVG rendering engine that allows you to display SVG files in picture field or variables. The **SVG EXPORT TO PICTURE** command can be used to generate a picture in 4D on the basis of an SVG description. Also note that the **GRAPH** command makes use of the integrated SVG engine of 4D.

For more information about this format, please refer to the following address: http://www.w3.org/Graphics/SVG/.

# SVG EXPORT TO PICTURE

SVG EXPORT TO PICTURE ( elementRef ; pictVar {; exportType} )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | Root XML element reference |
| pictVar | Picture | ⇒ | Picture variable to receive XML tree (SVG picture) |
| exportType | Longint | ⇒ | 0 = Do not store data source, 1 = Copy data source, 2 = Own data source (default) |

## Description

The **SVG EXPORT TO PICTURE** command saves an SVG format picture contained in an XML tree in the picture field or variable indicated by the *pictVar* parameter.

**Note:** For more information about the SVG format, refer to the **Overview of XML Utilities Commands** section.

Pass the root XML element reference that contains the SVG picture in *elementRef*.

Pass the name of the 4D picture field or variable that will contain the SVG picture in *pictVar*. The picture is exported in its native format (XML description) and is redrawn via the SVG rendering engine when it is displayed.

The optional *exportType* parameter specifies the way the XML data source is to be handled by the command. You can pass one of the following constants, found in the "**XML**" theme, in this parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Copy XML data source | Longint | 1 | 4D keeps a copy of the DOM tree with the picture, which means the picture can be saved in a picture field of the database and then redisplayed or exported at any time. |
| Get XML data source | Longint | 0 | 4D only reads the XML data source; it is not kept with the picture. This noticeably increases command execution speed; however, because the DOM tree is not kept, it is not possible to store or export the picture. |
| Own XML data source | Longint | 2 | 4D exports the DOM tree with the picture. The picture can be stored or exported and command execution is fast. However, the *elementRef* XML reference can then no longer be used by other 4D commands. This is the default mode for exporting when the *exportType* parameter is omitted. |

## Example

The following example can be used to display "Hello World" in a 4D picture:

```
C_PICTURE(vpict)
$svg:=DOM Create XML Ref("svg";"http://www.w3.org/2000/svg")
$ref:=DOM Create XML element($svg;"text";"font-size";26;"fill";"red")
DOM SET XML ATTRIBUTE($ref;"y";"1em")
DOM SET XML ELEMENT VALUE($ref;"Hello World")
SVG EXPORT TO PICTURE($svg;vpict;Copy XML data source)
DOM CLOSE XML($svg)
```

# SVG Find element ID by coordinates

SVG Find element ID by coordinates ( {* ;} pictureObject ; x ; y ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, pictureObject is an object name (string) If omitted, pictureObject is a field or variable |
| pictureObject | Picture | ⇒ | Object name (if * specified) or Field or Variable (if * omitted) |
| x | Longint | ⇒ | X coordinate in pixels |
| y | Longint | ⇒ | Y coordinate in pixels |
| Function result | String | ⇐ | ID of element found at the location X, Y |

## Description

The **SVG Find element ID by coordinates** command returns the ID ("id" or "xml:id" attribute) of the XML element found at the location set by the coordinates (x,y) in the SVG picture designated by the *pictureObject* parameter. This command can be used more particularly to create interactive graphic interfaces using SVG objects.

**Note:** For more information about the SVG format, refer to the **Overview of XML Utilities Commands** section.

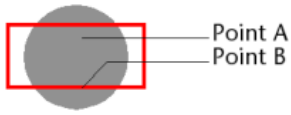
If you pass the optional * parameter, you indicate that the *pictureObject* parameter is an object name (string). If you do not pass this parameter, you indicate that the *pictureObject* parameter is a field or variable. In this case, you do not pass a string but a field or variable reference (field or variable object only).

Note that it is not mandatory for the picture to be displayed in a form. In this case, the "object name" type syntax is not valid and you must pass a field or variable name.

The coordinates passed in the *x* and *y* parameters must be expressed in pixels relative to the top left corner of the picture (0,0). In the context of a picture displayed in a form, you can use the values returned by the MouseX and MouseY system variables. These variables are updated in the On Clicked, On Double Clicked and On Mouse Up form events, as well as in the On Mouse Enter and On Mouse Move form events.

**Note:** In the picture coordinate system, MouseX and MouseY always specify the same point of the picture, regardless of the picture display format (except in the case of the "Replicated" format), even when the picture has been scrolled or zoomed.

The point taken into account is the first point reached. For example, in the case below, the command will return the ID of the circle if the coordinates of point A are passed and that of the rectangle if the coordinates of point B are passed:




When the coordinates correspond to superimposed or composite objects, the command returns the ID of the first object having a valid ID attribute by going back, if necessary, among the parent elements.

The command returns an empty string if:

- the root is reached without an "id" attribute having been found,
- the coordinates point does not belong to any object,
- the "id" attribute is an empty string.

**Note:** This command cannot detect objects whose opacity value ("fill-opacity" attribute) is less than 0.01.

## System variables and sets

If *pictureObject* does not contain a valid SVG picture, the command returns an empty string and the OK system variable is set to 0. Otherwise, if the command has been executed correctly, the OK system variable is set to 1.

# SVG Find element IDs by rect

SVG Find element IDs by rect ( {* ;} pictureObject ; x ; y ; width ; height ; arrIDs ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, pictureObject is an object name (string) If omitted, pictureObject is a variable |
| pictureObject | Picture | ⇒ | Object name (if * specified) or Field or variable (if * omitted) |
| x | Longint | ⇒ | Horizontal coordinate of top left corner of selection rectangle |
| y | Longint | ⇒ | Vertical coordinate of top left corner of selection rectangle |
| width | Longint | ⇒ | Width of selection rectangle |
| height | Longint | ⇒ | Height of selection rectangle |
| arrIDs | Text array | ⇐ | IDs of elements whose bounding rectangle intersects with the selection rectangle |
| Function result | Boolean | ⇐ | True = at least one element is found |

## Description

The **SVG Find element IDs by rect** command fills the Text or Alpha *arrIDs* array with the IDs ("id" or "xml:id" attribute) of the XML elements whose bounding rectangle intersects with the selection rectangle at the location specified by the *x* and *y* parameters.

The command returns True if at least one element is found (in other words if the *arrIDs* array is not empty), and False otherwise.

This command can be used in particular to manage interactive graphic interfaces.

If you pass the optional * parameter, you indicate that the *pictureObject* parameter is an object name (string). If you do not pass this parameter, you indicate that the *pictureObject* parameter is a field or a variable. In this case, you pass a field or variable reference (object field or variable only) instead of a string.

If you are working with a picture field or variable, the command uses the original picture, corresponding to the data source. However, if you are working with a form object, the command uses the current picture, that may have been modified via the **SVG SET ATTRIBUTE** command and that is kept with the properties of the form object.

The coordinates passed in the *x* and *y* parameters must be expressed in pixels in relation to the top left corner of the picture (0,0). You can use the values returned by the MouseX and MouseY **System Variables**. These variables are updated in the On Clicked and On Double Clicked form events as well as the in the On Mouse Enter and On Mouse Move form events.

**Note:** In the system of picture coordinates, [x;y] always specifies the same point, regardless of the picture display format, apart from the "Replicated" format.

All elements whose bounding rectangle intersects with the selection rectangle are taken into account, even those that are under other elements.

# SVG GET ATTRIBUTE

SVG GET ATTRIBUTE ( {* ;} pictureObject ; element_ID ; attribName ; attribValue )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, pictureObject is an object name (string) If omitted, pictureObject is a variable |
| pictureObject | Picture | ⇒ | Object name (if * specified) or Variable or field (if * omitted) |
| element_ID | Text | ⇒ | ID of element whose attribute value you want to get |
| attribName | String | ⇒ | Attribute whose value you want to get |
| attribValue | String, Longint | ⇐ | Current value of attribute |

## Description

The **SVG GET ATTRIBUTE** command is used to get the current value of the *attribName* attribute in an object or an SVG picture.

If you pass the optional * parameter, you indicate that the *pictureObject* parameter is an object name (string). In this case, the command returns the value of the attribute for the rendered image attached to the object. This value may have been modified by **SVG SET ATTRIBUTE** for example.
If you do not pass the * parameter, you indicate that the *pictureObject* parameter is a variable or a field. Therefore, you pass a variable (object variable only) or field reference instead of a string. In this case, the command returns the value of the attribute for the initial rendered image (corresponding to the data source of the variable).

**Note:** This principle also applies to the **SVG Find element ID by coordinates** command.

The *element_ID* parameter is used to set the ID ("id" or "xml:id" attribute) of the element whose attribute value you want to get.

For more information about SVG attributes, please refer to the description of the **SVG SET ATTRIBUTE** command. Here is a list of 4D attributes reserved and dedicated to animation:

| Attributes | Access | Comments |
|---|---|---|
| 4D-text | read/write | Replaces/reads the contents of the text node. Can be used with 'text' 'tspan' and 'textArea' elements. |
| 4D-bringToFront | write | If 'true', move node in front of sibling nodes. Can only be used with the **SVG SET ATTRIBUTE** command. |
| 4D-isOfClass-{IDENT [[S\|COMMA] IDENT]*} | read | Returns 'true' if inherited class attribute of node contains all class name(s); otherwise, returns 'false'. Returns for example true for "4D-isOfClass-land" if the inherited class of the node is "land department01"). |
| 4D-enableD2D | read/write | If 'false', disables Direct2D for the SVG rendering engine. In fact, SVG filters are not rendered in Direct2D but they are in GDI/GDIPlus. This option lets you use SVG filters even when the database is in Direct2D. Note that this option is only taken into account when a picture has already been loaded into the *pictureObject*. However, since this option is applied globally to the engine, you only need to set it once per session (for example with a simple SVG loaded in memory from a text variable when the database is started). |

# SVG SET ATTRIBUTE

SVG SET ATTRIBUTE ( {* ;} pictureObject ; element_ID ; attribName ; attribValue {; attribName2 ; attribValue2 ; ... ; attribNameN ; attribValueN} {; *})

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, pictureObject is an object name (string) If omitted, pictureObject is a variable |
| pictureObject | Picture | ⇒ | Object name (if * specified) or Variable or field (if * omitted) |
| element_ID | Text | ⇒ | ID of element where one or more attributes are set |
| attribName | String | ⇒ | Attribute to be specified |
| attribValue | String, Longint | ⇒ | New value of attribute |
| * | Operator | ⇒ | If passed = modify internal DOM tree of SVG image (variable only) |

## Description

The **SVG SET ATTRIBUTE** command is used to modify the value of an existing attribute in the SVG rendering tree of a displayed image or in the internal DOM tree of an image.

If you pass the optional * parameter, you indicate that the *pictureObject* parameter is an object name (string). In this case, the command applies to the parameters of the rendered image attached to the object (note that the parameters and therefore the rendered image of the object are only created if the **SVG SET ATTRIBUTE** command is called at least once). If you do not pass the * parameter, you indicate that the *pictureObject* parameter is a variable or a field. Therefore, you pass a variable (object variable only) or field reference instead of a string. In this case, the command applies to the rendered images of all the objects that use the variable or the field.

By default, modifications made by this command apply only to the rendered images; they are not stored in the data source (internal DOM tree) and are lost when the picture is erased by programming or when the form is closed. However, you can transfer these changes into the internal DOM tree of the image when the *pictureObject* parameter references a variable: you just need to pass the second * as the last parameter. This lets you keep the modifications made on the fly.

**Notes:**

- Transferring modifications into the internal DOM tree is not possible when the *pictureObject* parameter references an object.
- To be able to transfer modifications, the SVG variable must have been created from a DOM document (with **DOM EXPORT TO VAR**). If the SVG variable was created from a file, when you pass the second * parameter, the command does nothing and an error is generated because, in this case, the data source does not contain a modifiable DOM document.
- To change the data source of an SVG image, you can also use the **XML DOM** commands or the **MissingRef** provided by 4D.

The *element_ID* parameter is used to specify the ID ("id" or "xml:id" attribute) of the element whose attribute(s) you want to modify.

In the *attribName* and *attribValue* parameters, pass, respectively, the attribute to set and its value (as variables, fields or literal values). You can pass as many attribute/value pairs as you want.

The **SVG SET ATTRIBUTE** command is used to modify (but not to add or delete) most of the SVG attributes, such as, for instance, 'fill', 'opacity', 'font-family', and so on. For a complete definition of the SVG attributes, please refer to the reference documents available on the Internet, for example: http://www.w3.org/TR/SVG11/attindex.html. The rendered image is updated immediately; the modifications are transferred on to the child elements for inherited styles.

Note that for technical reasons, the attributes of certain elements as well as certain attributes cannot be modified. The following table lists the elements that can be modified, and those that cannot, as well as the attributes that cannot be modified:

**Elements whose attributes can be modified**

| | |
|---|---|
| svg | Restrictions :<br>- "width" and "height" cannot be modified(1)<br>- "viewBox" can only be modified if "width" and "height" are specified in the original document |
| g | |
| defs | |
| use | |
| filter | Restriction: fe_xxx child elements cannot be modified |
| circle | |
| ellipse | |
| line | |
| polyline | |
| polygon | |
| path | |
| rect | |
| text, tspan, textArea | The specific "4d-text" attribute is used to modify the text of a "text", "tspan" or "textArea" element (see the example) |
| Image | |

**Elements whose attributes cannot be modified**

| | |
|---|---|
| linearGradient, radialGradient, Stop, solidColor, marker, symbol, clipPath, filter et les éléments commençant par fe, style, pattern | This group designates all the elements that can be referenced or contained in an element that can be referenced. This means that it is not possible, for example, to redefine the attributes of a gradient (but it is possible to change the gradient used). Similarly, to change a black color marker to a red marker, it is necessary to define both markers in the SVG document (one black and one red) and to select one of them. It is not possible either for example to modify the color of a rectangle if its parent is a symbol or marker element |

**Attributes that cannot be modified**

| | |
|---|---|
| id or xml:id | |
| lang or xml:lang | |
| class or xml:class | |
| width, height | Concerns the attributes of the 'svg' element only(1) |

(1) These attributes cannot be modified because they define and structure the resulting image. The *width* and *height* attributes of the *svg* element are used to define the initial dimensions of the picture in 4D and these dimensions must remain constant after the picture is created (it is however possible to modify the dimensions of the resulting picture with the **TRANSFORM PICTURE** command of 4D).

You can also refer to the description of the **SVG GET ATTRIBUTE** command to see the list of 4D attributes that are reserved and dedicated to animation.

If you attempt to modify the attribute of an element that is not supported or one of its child elements, the command does nothing and no error is generated.

If the command is not executed in the context of a form or if an invalid *pictureObject* is passed, the *OK* variable is set to 0. If the command has been executed correctly, it is set to 1.

## Example

Modification of the contents of a Text type element:

```
SVG SET ATTRIBUTE(*;picture_object_name;text_element_ID;"4d-text";"This is a text")
```

**Note:** There is no namespace in order that the attribute could be used in a CSS style sheet without risk of conflict.

# SVG SHOW ELEMENT

SVG SHOW ELEMENT ( {* ;} pictureObject ; id {; margin} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, pictureObject is an object name (string) If omitted, pictureObject is a variable |
| pictureObject | Picture | ⇒ | Object name (if * specified) or Variable or field (if * omitted) |
| id | Text | ⇒ | ID attribute of element to display |
| margin | Longint | ⇒ | Margin of visibility (in pixels by default) |

## Description

The **SVG SHOW ELEMENT** command moves the *pictureObject* SVG document in order to show the element whose "id" attribute is specified by the *id* parameter.

If you pass the optional * parameter, you indicate that the *pictureObject* parameter is an object name (string). In this case, the command applies to the rendered picture attached to the object. If you do not pass this parameter, you indicate that the *pictureObject* parameter is a field or a variable and you pass a variable (object variable only) or field reference instead of a string. In this case, the command applies to the rendered pictures of all the objects that use the variable (but not the initial rendered picture).

The command moves the SVG document so that all of the object, whose limits are defined by its bounding rectangle, is visible. The *margin* parameter is used to configure the amplitude of the movement by specifying the distance that must separate the object displayed from the borders of the document. In other words, the bounding rectangle will be increased by *margin* pixels in width and in height. By default, the movement value is 4 pixels.

This command only takes effect in "top left" display mode (with scrollbars).

If this command is not executed in the context of a form or if an invalid *pictureObject* is passed, the *OK* variable is set to 0. If the command is executed correctly, it is set to 1.

# System Documents

- System Documents
- Append document
- CLOSE DOCUMENT
- Convert path POSIX to system
- Convert path system to POSIX
- COPY DOCUMENT
- CREATE ALIAS
- Create document
- CREATE FOLDER
- DELETE DOCUMENT
- DELETE FOLDER
- Document creator
- DOCUMENT LIST
- Document to text
- Document type
- FOLDER LIST
- GET DOCUMENT ICON
- Get document position
- GET DOCUMENT PROPERTIES
- Get document size
- Get localized document path
- MAP FILE TYPES
- MOVE DOCUMENT
- Open document
- RESOLVE ALIAS
- Select document
- Select folder
- SET DOCUMENT CREATOR
- SET DOCUMENT POSITION
- SET DOCUMENT PROPERTIES
- SET DOCUMENT SIZE
- SET DOCUMENT TYPE
- SHOW ON DISK
- Test path name
- TEXT TO DOCUMENT
- VOLUME ATTRIBUTES
- VOLUME LIST

# System Documents

## Introduction

All the documents and applications you use on your computer are stored as **files** on the hard disk(s) **connected** to or **mounted** on your machine, or floppy disk(s) or other similar permanent storage devices. Within 4D, we use the terms **file** or **document** to refer to these documents and applications. However, most commands in this theme use the term "document" because most of the time you will use them to access documents (rather than application or system files) on disk.

A hard disk can be formatted as one or several partitions, each of which is called a **volume**. It does not matter if two volumes are physically present on the same hard disk; at the 4D First level, you will usually treat these volumes as separate and equal entities.

A volume can be located on a hard disk physically connected to your machine or mounted over the network through a file sharing protocol such as TCP/IP, AFP ou SMB (Macintosh). Whatever the case, when using the System Documents commands at the 4D level, you treat all these volumes in the same way (unless you know what you are doing and use Plug-ins to extend the capability of your application in that domain).

Each volume has a **volume name**. On Windows, volumes are designated by a letter followed by a colon. Usually C: and D: are used to designate the volumes you use for booting your system (unless you configure your PC otherwise). Then the letters E: through Z: are used for the additional volumes connected or mounted to your PC (DVD drives, additional drives, network drives, and so on). On Macintosh, volumes have natural names; these are the names you see on the desktop at the Finder level.

Normally, you classify your documents into **folders**, which themselves can contain other folders. It is not a good idea to accumulate hundreds or thousands of files at the same level of a volume; it is messy and it slows down your system. On Windows, a folder is (or was) called a **directory**. Folders have always been called so on the Macintosh.

To uniquely identify a document, you need to know the name of the volume and the name(s) of the folder(s) where the document is located as well as the name of the document itself. If you concatenate all these names, you get the **pathname** to the document. Within this pathname, folder names are separated by a special character called the **folder separator** symbol. On Windows, this character is the backslash (*¥)*; on Macintosh it is the colon (*:*).

Let's look at an example. You have a document **Important Memo** located in the **Memos** folder, which is located in the **Documents** folder, which is located in the **Current Work** folder.

On Windows, if the whole thing is located on the C: drive (volume), the pathname of the document is:

C:¥Current Work¥Documents¥Memos¥Important Memo.TXT

**Note:** The ¥ character is also used by the method editor of 4D to designate escape sequences. In order to avoid any interpretation problems, the editor automatically transforms pathnames such as C:¥Disk into C:¥¥Disk. For more information, refer to the paragraph below titled “Specifying Document names or Document pathnames”.

On Macintosh, if the whole thing is located on the disk (volume) **Internal Drive**, the pathname of the document is:

**Internal Drive:Current Work:Documents:Memos:Important Memo**

On Windows, the name of the document is suffixed with *.TXT*; we will see why in the next section.

Whatever the platform, the full pathname of a document can be expressed as follows:

VolName DirSep { DirName DirSep { DirName DirSep { ... } } } DocName

All the documents (files) located on volumes have several characteristics, usually called **attributes** or **properties**: the **name** of the document itself, the **type** and the **creator**.

## DocRef: Document reference number

A document is **open in read/write mode**, **open in read-only mode** or **closed**. Using the built-in 4D commands, a document can be opened in read/write mode by only one process at a time. One process can open several documents, several processes can open multiple documents, you can open the same document in read-only mode as many times as necessary, but you cannot open the same document in read/write mode twice at a time.

You open a document with the **Open document**, **Create document** and **Append document** commands. The **Create document** and **Append document** commands automatically open documents in read/write mode. Only the **Open document** command lets you choose the opening mode. Once a document is open, you can read and write characters from

and to the document (see the **RECEIVE PACKET** and **SEND PACKET** commands). When you are finished with the document, you usually close it using the **CLOSE DOCUMENT** command.

All open documents are referred to using the **DocRef** expression returned by the **Open document**, **Create document** and **Append document** commands. A DocRef uniquely identifies an open document. It is formally an expression of the Time type. All commands working with open documents expect DocRef as a parameter. If you pass an incorrect DocRef to one of these commands, a file manager error occurs.

**Note:** When it is called from a preemptive process, a *DocRef* reference can only be used from this preemptive process. When it is called from a cooperative process, a *DocRef* reference can be used from any other cooperative process.

## Handling I/O errors

When you access (open, close, delete, rename, copy) documents, when you change the properties of a document or when you read and write characters in a document, I/O errors may occur. A document might not be found; it may be locked; it may be already open in write mode. You can catch these errors with an error-handling method installed with **ON ERR CALL**. Most of the errors that can occur while using system documents are described in the section **OS File Manager Errors (-124 -> -33)**.

## The Document system variable

The commands **Open document**, **Create document**, **Append document** and **Select document** enable you to access a document using the standard Open or Save file dialog boxes. When you access a document through a standard dialog, 4D returns the full pathname of the document in the *Document* system variable. This system variable has to be distinguished from the *document* parameter that appears in the parameter list of the commands.

## Specifying Document names or Document pathnames

Most of the routines of this section expecting a document name accept both a name or a pathname to the document (except when signaled otherwise). If you pass a name, the command looks for the document within the folder of the database. If you pass a pathname, it must be valid.

If you pass a wrong name or a wrong pathname, the command generates a file manager error that you can intercept using an **ON ERR CALL** method.

### Entering Windows pathnames and escape sequences

The method editor of 4D allows the use of escape sequences. An escape sequence is a set of characters that are used to replace a “special” character. The sequence begins with a backslash ¥, followed by a character. For example, *¥t* is the escape sequence for the Tab character.

The ¥ character is also used as the separator in pathnames under Windows. In general, 4D will correctly interpret Windows pathnames that are entered in the method editor by replacing single backslashes *¥* with double backslashes *¥¥*. For example, **C:¥Folder** will become **C:¥¥Folder**.

However, if you write **C:¥MyDocuments¥New**, 4D will display **C:¥¥MyDocuments¥New**. In this case, the second ¥is incorrectly interpreted as *¥N* (an existing escape sequence). You must therefore enter a double *¥¥* when you want to insert a backslash before a character that is used in one of the escape sequences recognized by 4D.

The following escape sequences are recognized by 4D:

| Escape sequence | Character replaced |
|---|---|
| ¥n | LF (New line) |
| ¥t | HT (Horizontal tab) |
| ¥r | CR (Carriage return) |
| ¥¥ | ¥ (Backslash) |
| ¥" | " (Quotes) |

## Absolute or relative pathname

Most commands for managing 4D documents and folders accept either relative or absolute pathnames:

- **Relative pathnames** define a location with respect to a folder located on disk. In 4D, a relative pathname is usually expressed with respect to the database folder, i.e. the folder containing the structure file. Relative pathnames are

especially useful when deploying applications in heterogenous environments.
- **Absolute pathnames** define a location with respect to the root of the volume and so they do not depend on the current location of the database folder.

To determine whether a pathname passed to a command must be interpreted as absolute or relative, 4D applies a specific algorithm on each platform.

### Under Windows

If the parameter contains only two characters **and** if the second one is a ':',
**or** if the text contains ':' and '¥' as the second and third character,
**or** if the text starts with "¥¥",
then the pathname is **absolute**.

In all other cases, the pathname is **relative**.

Examples with the **CREATE FOLDER** command:

```
CREATE FOLDER("lundi") // relative path
CREATE FOLDER("¥Monday") // relative path
CREATE FOLDER("¥Monday¥Tuesday") // relative path
CREATE FOLDER("c:") // absolute path
CREATE FOLDER("d:¥Monday") // absolute path
CREATE FOLDER("¥¥srv-Internal¥temp") // absolute path
```

### Under Mac OS

If the text starts with a folder separator ':',
**or** if does not contain any,
then the path is **relative**.

In all other cases, it is **absolute**.

Examples with the **CREATE FOLDER** command:

```
CREATE FOLDER("Monday") // relative path
CREATE FOLDER("macintosh hd:") // absolute path
CREATE FOLDER("Monday:Tuesday") // absolute path (a volume must be called Monday)
CREATE FOLDER(":Monday:Tuesday") // relative path
```

## Useful Project Methods when handling documents on disk

**• Detecting on which platform you're running**

Although 4D provides commands, such as **MAP FILE TYPES**, for eliminating coding variations due to platform specificities, once you start to work at a lower level when handling documents on disk (such as programmatically obtaining pathnames), you need to know if you are running on a Macintosh or a Windows platform.

The ***On Windows*** project method listed here tells whether your database is running on Windows:

```
//On Windows Project Method
//On Windows -> Boolean
//On Windows -> True if on Windows

C_BOOLEAN($0)
C_LONGINT($vlPlatform;$vlSystem;$vlMachine)

PLATFORM PROPERTIES($vlPlatform;$vlSystem;$vlMachine)
$0:=($vlPlatform=Windows)
```

**• Extracting the file name from a long name**

Once you have obtained the long name (pathname + file name) of a document, you may need to extract the file name of the document from that long name in order, for example, to display it in the title of a window. The ***Long name to file name*** project method does this on both Windows and Macintosh.

```
//Long name to file name Project Method
//Long name to file name ( String ) -> String
//Long name to file name ( Long file name ) -> File name
```

```
C_TEXT($1;$0)
C_LONGINT($viLen;$viPos;$viChar;$viDirSymbol)

$viDirSymbol:=Character code(Folder separator)
$viLen:=Length($1)
$viPos:=0
For($viChar;$viLen;1;-1)
   If(Character code($1≤$viChar≥)=$viDirSymbol)
      $viPos:=$viChar
      $viChar:=0
   End if
End for
If($viPos>0)
   $0:=Substring($1;$viPos+1)
Else
   $0:=$1
End if
If(<>vbDebugOn) //Set this variable to True or False in the On Startup database method
   If($0="")
      TRACE
   End if
End if
```

**• Extracting the pathname from a long name**

Once you have obtained the long name (pathname + file name) of a document, you may need to extract the pathname of the directory where the document is located from that long name; for instance, you may want to save additional documents at the same location. The ***Long name to path name*** project method does this on both Windows and Macintosh.

```
 //Long name to path name Project Name
 //Long name to path name ( String ) -> String
 //Long name to path name ( Long file name ) -> Path name

C_TEXT($1;$0)
C_LONGINT($viLen;$viPos;$viChar;$viDirSymbol)

$viDirSymbol:=Character code(Folder separator)
$viLen:=Length($1)
$viPos:=0
For($viChar;$viLen;1;-1)
   If(Character code($1≤$viChar≥)=$viDirSymbol)
      $viPos:=$viChar
      $viChar:=0
   End if
End for
If($viPos>0)
   $0:=Substring($1;1;$viPos)
Else
   $0:=$1
End if
If(<>vbDebugOn) //Set this variable to True or False in the On Startup database method
   If($0="")
      TRACE
   End if
End if
```

# Append document

Append document ( document {; fileType} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name or Full document pathname or Empty string for standard file dialog box |
| fileType | String | ⇒ | List of types of documents to be screened, or "*" to not screen the documents |
| Function result | DocRef | ⇐ | Document reference number |

## Description

The **Append document** command does the same as thing as **Open document**: it opens a document on disk.

The only difference is that **Append document** sets the file position at the end of the document while **Open document** sets its at the beginning of the document.

Refer to **Open document** for more details about using **Append document**.

## Example

The following example opens an existing document called Note, appends the string “and so long” and a carriage return onto the end of the document, and closes the document. If the document already contained the string “Good-bye”, the document would now contain the string “Good-bye and so long”, followed by a carriage return:

```
C_TIME(vhDocRef)
vhDocRef:=Append document("Note.txt") ` Open Note document
SEND PACKET(vhDocRef;" and so long"+Char(13)) ` Append a string
CLOSE DOCUMENT(vhDocRef) ` Close the document
```

# CLOSE DOCUMENT

CLOSE DOCUMENT ( docRef )

| Parameter | Type | | Description |
|---|---|---|---|
| docRef | DocRef | ➡ | Document reference number |

## Description

**CLOSE DOCUMENT** closes the document specified by *docRef*.

Closing a document is the only way to ensure that the data written to a file is saved. You must close all the documents you open with the commands **Open document**, **Create document** or **Append document**.

## Example

The following example lets the user create a new document, writes the string “Hello” into it, and closes the document:

```
C_TIME(vhDocRef)
vhDocRef:=Create document("")
If(OK=1)
   SEND PACKET(vhDocRef;"Hello") ` Write one word into the document
   CLOSE DOCUMENT(vhDocRef) ` Close the document
End if
```

# Convert path POSIX to system

Convert path POSIX to system ( posixPath {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| posixPath | Text | ⇒ | POSIX pathname |
| * | Operator | ⇒ | Encoding option |
| Function result | Text | ⇐ | Pathname expressed in system syntax |

## Description

The **Convert path POSIX to system** command converts a pathname expressed with the POSIX (Unix) syntax into a pathname expressed with the system syntax.

Pass the complete pathname of a file or folder in the *posixPath* parameter, expressed with the POSIX syntax. This path must be absolute (it must begin with the "/" character). You must pass a disk path; it is not possible to pass a network path (beginning, for example, with ftp://ftp.mysite.fr).

The command returns the complete pathname of the file or folder expressed in the current system syntax.

The optional * parameters can be used to indicate whether the *posixPath* parameter is encoded. If this is the case, you must pass this parameter, otherwise the conversion will not be valid. The command returns the pathname without encoding.

## Example 1

Examples under Mac OS:

```
$path:=Convert path POSIX to system("/Volumes/machd/file 2.txt")
 //returns "machd:file 2.txt"
$path:=Convert path POSIX to system("/Volumes/machd/file%202.txt";*)
 //returns "machd:file 2.txt"
$path:=Convert path POSIX to system("/file 2.txt")
 //returns "machd:file 2.txt" if machd is the startup disk
```

## Example 2

Examples under Windows:

```
$path:=Convert path POSIX to system("c:/docs/file 2.txt")
 //returns "c:¥¥docs¥¥file 2.txt"
$path:=Convert path POSIX to system("c:/docs/file%202.txt";*)
 //returns "c:¥¥docs¥¥file 2.txt"
```

# Convert path system to POSIX

Convert path system to POSIX ( systemPath {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| systemPath | Text | ⇒ | Relative or absolute pathname expressed in system syntax |
| * | Operator | ⇒ | Encoding option |
| Function result | Text | ⇐ | Chemin d'accès absolu exprimé en syntaxe POSIX |

## Description

The **Convert path system to POSIX** command converts a pathname expressed with the system syntax as a pathname expressed with the POSIX (Unix) syntax.

Pass the pathname for a file or folder in the *systemPath* parameter, expressed with the system syntax (Mac OS or Windows). This path may be absolute or relative to the database folder (folder containing the database structure file). It is not mandatory that the elements of the path actually exist on the disk at the time the command is executed (the command does not test the validity of the pathname).

The command returns the complete pathname of the file or folder expressed in the POSIX syntax. The command always returns an absolute pathname, regardless of the type of path passed in *systemPath*. If you passed a relative pathname in *systemPath*, 4D completes the value returned by adding the pathname of the database folder.

The optional * parameter can be used to specify the encoding of the POSIX path. By default, **Convert path system to POSIX** does not encode the special characters of the POSIX path. If you pass the * parameter, the special characters are translated (for example, "My folder" becomes "My%20folder").

## Example 1

Examples under OS X

```
$path:=Convert path system to POSIX("machd:file 2.txt")
 //machd is the startup disk
 //returns "/file 2.txt"
$path:=Convert path system to POSIX("disk2:file 2.txt")
 //disk2 is an additional disk (not the startup)
 //returns "/Volumes/disk2/file 2.txt"
$path:=Convert path system to POSIX("machd:file 2.txt";*)
 //returns "/file%202.txt"
$path:=Convert path system to POSIX(":resources:images") //relative path
 //returns "/User/mark/Documents/videodatabase/resources/images"
$path:=Convert path system to POSIX("resources:images") //absolute path
 //returns "/resources/images"
```

## Example 2

Example under Windows

```
$path:=Convert path system to POSIX("c:¥docs¥file 2.txt")
 `returns "c:/docs/file 2.txt"
$path:=Convert path system to POSIX("¥¥srv¥tempo¥file.txt")
 `returns "//srv/tempo/file.txt"
```

# COPY DOCUMENT

COPY DOCUMENT ( sourceName ; destinationName {; newName} {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| sourceName | String | ➡ | Pathname of file or folder to be copied |
| destinationName | String | ➡ | Name or pathname of copied file or folder |
| newName | String | ➡ | New name of copied file or folder |
| * | Operator | ➡ | Override existing document if any |

## Description

The **COPY DOCUMENT** command copies the file or folder specified by *sourceName* to the location specified by *destinationName* and, optionally, renames it.

- **Copying files**
  In this case, the *sourceName* parameter can contain:
  - either a complete file pathname expressed with respect to the root of the volume,
  - or a pathname relative to the database folder.

  The *destinationName* parameter can contain several types of locations:
  - a complete file pathname expressed with respect to the root of the volume: the file is copied to this location
  - a file name or file pathname relative to the database folder: the file is copied into the database folder (the subfolders must already exist)
  - a complete folder pathname or a pathname relative to the database folder (*destinationName* must end with the folder separator for the platform): the file is copied into the designated folder. These folders must already exist on the disk; they are not created.

  An error is generated if there is already a document named *destinationName* unless you specify the optional * parameter which, in this case, instructs **COPY DOCUMENT** to delete and override the existing document in the destination location.

- **Copying folders**
  To indicate that you are designating a folder, the strings passed in *sourceName* and *destinationName* must end with a folder separator for the platform. For example, under Windows "C:¥¥Element¥¥" designates a folder and "C:¥¥Element" designates a file.
  To copy a folder, pass its complete pathname in *sourceName*. This folder must already exist on the disk. When a folder is set in the *sourceName* parameter, a folder must also be designated in the *destinationName* parameter. You must pass the complete folder pathname (where each element must already exist on the disk)
  If a folder with the same name as the one designated by the *sourceName* parameter already exists at the location set by the *destinationName* parameter and it is not empty, 4D checks its contents before copying the items. An error is generated when a file with the same name already exists, unless you have passed the optional * parameter which, in this case, indicates to the command to delete and replace the file in the destination location. .

Note that you can pass a file in the *sourceName* parameter and a folder in the *destinationName* parameter, in order to copy a file into a folder.

The optional *newName* parameter, when it is passed, renames the document copied to its destination location (file or folder). When it is passed in the context of copying a file, this parameter replaces the name (if any) passed in the *destinationName* parameter.

## Example 1

The following example duplicates a document in its own folder:

```
COPY DOCUMENT("C:¥¥FOLDER¥¥DocName";"C:¥¥FOLDER¥¥DocName2")
```

## Example 2

The following example copies a document to the database folder (provided **C:¥¥FOLDER** is not the database folder):

```
COPY DOCUMENT("C:¥¥FOLDER¥¥DocName";"DocName")
```

## Example 3

The following example copies a document from one volume to another one:

```
COPY DOCUMENT("C:¥¥FOLDER¥¥DocName";"F:¥¥Archives¥¥DocName.OLD")
```

## Example 4

The following example duplicates a document in its own folder overriding an already existing copy:

```
COPY DOCUMENT("C:¥¥FOLDER¥¥DocName";"C:¥¥FOLDER¥¥DocName2";*)
```

## Example 5

Copying a file into a specific folder while keeping the same name:

```
COPY DOCUMENT("C:¥¥Projects¥¥DocName";"C:¥¥Projects¥¥")
```

## Example 6

Copying a file into a specific folder while keeping the same name and overriding the existing document:

```
COPY DOCUMENT("C:¥¥Projects¥¥DocName";"C:¥¥Projects¥¥"; *)
```

## Example 7

Copying a folder into another folder (both folders must already be present on the disk):

```
COPY DOCUMENT("C:¥¥Projects¥¥";"C¥¥Archives¥¥2011¥¥")
```

## Example 8

The following examples create different files and folders in the database folder (examples under Windows). In each case, the "folder2" folder must exist:

```
COPY DOCUMENT("folder1¥¥name1";"folder2¥¥")
 //creates the "folder2/name1" file

COPY DOCUMENT("folder1¥¥name1";"folder2¥¥" ; "new")
 //creates the "folder2/new" file

COPY DOCUMENT("folder1¥¥name1";"folder2¥¥name2")
 //creates the "folder2/name2" file

COPY DOCUMENT("folder1¥¥name1";"folder2¥¥name2";"new")
 //creates the "folder2/new" file (name2 is ignored)

COPY DOCUMENT("folder1¥¥" ; "folder2¥¥")
 //creates the "folder2/folder1/" folder

COPY DOCUMENT("folder1¥¥" ; "folder2¥¥" ; "new")
 //creates the "folder2/new/" folder
```

# CREATE ALIAS

CREATE ALIAS ( targetPath ; aliasPath )

| Parameter | Type | | Description |
|---|---|---|---|
| targetPath | String | ⇒ | Name or access path of the alias/shortcut target |
| aliasPath | String | ⇒ | Name or full pathname for the alias or shortcut |

## Description

The **CREATE ALIAS** command creates an alias (named “shortcut” under Windows) for the target file or folder passed in *targetPath*. The name and location are defined by the *targetPath* parameter.

An alias can be made for any kind of document or folder. The alias icon will be the same as the target item. The icon contains a small arrow at the bottom left side. Under Mac OS, the icon name is also displayed in italics characters.

This command does not assign a name by default, the name has to be passed in the *aliasPath* parameter. If just a name is passed in this parameter, the alias is created in the current working folder (usually the folder containing the structure file).

**Note:** Under Windows, the shortcuts are designated by a “.LNK” extension (invisible). If this extension is not passed, it is automatically added by the command.

If an empty string is passed in the *targetPath*, the command does nothing.

## Example

Your database generates text files called “Report Number” sorted in the database folder. The user would like to create shortcuts to these reports and to store them at a convenient location:

```
  `Method CREATE_REPORT
C_TEXT($vtRport)
C_STRING(250;$vtpath)
C_STRING(80;$vaname)
C_TIME(vDoc)
C_INTEGER($ReportNber)

$vTReport:=... `Create report
$ReportNber:=... `Compute the report number
$vaName:="Report"+String($ReportNber)+".txt" `File name
vDoc:=Create document($vaName)
If(OK=1)
   SEND PACKET(vDoc;$vTReport)
   CLOSE DOCUMENT(vDoc)
 `Add the alias
   CONFIRM("Create an alias for this report?")
   If(OK=1)
      $vtPath:=Select folder("Where do you want the alias to be created?")
      If(OK=1)
         CREATE ALIAS($vaName;$vtPath+$vaName)
         If(OK=1)
            SHOW ON DISK($vtPath+$vaName)
 `Show the alias location
         End if
      End if
   End if
End if
```

## System variables and sets

The OK system variable is set to 1 if the command execution was successful; otherwise it is set to 0.

# Create document

Create document ( document {; fileType} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name or Full document pathname or Empty string for standard file dialog box |
| fileType | String | ⇒ | List of types of documents to be screened, or "*" to not screen the documents |
| Function result | DocRef | ⇐ | Document reference number |

## Description

The**Create document** command creates a new document and returns its document reference number.

Pass the name or full pathname of the new document in *document*. If *document* already exists on the disk, it is overwritten. However, if *document* is locked or already open, an error is generated.

If you pass an empty string in *document*, the Save As dialog box appears and you can then enter the name of the document you want to create. If you cancel the dialog, no document is created; **Create document** returns a null DocRef and sets the OK variable to 0.

If the document is correctly created and opened, **Create document** returns its document reference number and sets the OK variable to 1. The system variable Document is updated and returns the complete access path of the created document.

Whether or not you use the Save As dialog box, **Create document** creates a .TXT (Windows) or TEXT (Macintosh) document by default. If you want to create another type of document, pass the *fileType* parameter.

In the *fileType* parameter, pass the type(s) of file(s) that can be selected in the opening dialog box. You can pass a list of several types separated by a ; (semi-colon). For each type set, an item will be added to the menu used for choosing the type in the dialog box.

Under Mac OS, you can pass either a standard Mac OS type (TEXT, APPL, etc.), or a UTI (Uniform Type Identifier) type. UTIs are defined by Apple in order to meet standardization needs for file types. For example, "public.text" is the UTI type of text type files. For more information about UTIs, refer to the following address: https://developer.apple.com/library/mac/#documentation/Miscellaneous/Reference/UTIRef/Articles/System-DeclaredUniformTypeIdentifiers.html.

Under Windows, you can also pass a standard Mac OS file type — 4D makes the correspondence internally — or file extensions (.txt, .exe, etc.). Note that under Windows, the user can “force” the display of all file types by entering *.* in the dialog box. However, in this case, 4D will carry out an additional check of the selected file types: if the user selects an unauthorized file type, the command returns an error.

If you do not want to restrict the displayed files to one or more types, pass the "*" (star) string or ".*" in *fileType*.

On Windows you pass a Windows file extension or Macintosh file type mapped through the **MAP FILE TYPES** mechanism. If you want to create a document without an extension, a document containing several extensions, or a document containing an extension with more than three characters, do not use the *type* parameters and pass the full name in *document* (see example2).

Once you have created and opened a document, you can write and read the document using the **RECEIVE PACKET** and **SEND PACKET** commands that you can combine with the **Get document position** and **SET DOCUMENT POSITION** commands in order to directly access any part of the document.

Do not forget to eventually call **CLOSE DOCUMENT** for the document.

## Example 1

The following example creates and opens a new document called Note, writes the string “Hello” into it, and closes the document:

```
C_TIME(vhDoc)
vhDoc:=Create document("Note.txt") ` Create new document called Note
If(OK=1)
   SEND PACKET(vhDoc;"Hello") ` Write one word in the document
   CLOSE DOCUMENT(vhDoc) ` Close the document
End if
```

## Example 2

The following example creates documents with non-standard extensions under Windows:

```
$vtMyDoc:=Create document("Doc.ext1.ext2") `Several extensions
$vtMyDoc:=Create document("Doc.shtml") `Long extension
$vtMyDoc:=Create document("Doc.") `No extension (the period "." is mandatory)
```

## System variables and sets

If the document has been created correctly, the system variable OK is set to 1 and the system variable Document contains the full pathname and the name of *document*.

# CREATE FOLDER

CREATE FOLDER ( folderPath {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| folderPath | String | ⇒ | Pathname to new folder to create |
| * | Operator | ⇒ | Create folder hierarchy |

## Description

The **CREATE FOLDER** command creates a folder according to the pathname you pass in *folderPath*.

If you pass a name in *folderPath*, the folder is created in the database folder.

In *folderPath*, you can also pass a folder hierarchy starting from the root of the volume or from the database folder (in this case, the string must end with a folder separator).
If you omit the * parameter, an error is generated and no folder is created if any one of the intermediary folders does not exist.
If you pass the * parameter, **CREATE FOLDER** recreates the folder hierarchy if necessary and no error is generated. In this case, you can also pass a document pathname in *folderPath*. The document name is then ignored but the folder hierarchy specified in *folderPath* is created recursively

## Example 1

The following example creates the “Archives” folder in the folder of the database:

```
CREATE FOLDER("Archives")
```

## Example 2

The following example creates the Archives folder in the folder of the database, then it creates the “January” and “February” subfolders:

```
CREATE FOLDER("Archives")
CREATE FOLDER("Archives¥¥January")
CREATE FOLDER("Archives¥¥February")
```

## Example 3

The following example creates the “Archives” folder at the root level of the C volume:

```
CREATE FOLDER("C:¥¥Archives")
```

## Example 4

Creation of the "C:¥Archives¥2011¥January¥" folder hierarchy:

```
CREATE FOLDER("C:¥¥Archives¥¥2011¥¥January¥¥";*)
```

## Example 5

Creation of the "¥February¥" subfolder in the existing "C:¥Archives¥" folder:

```
CREATE FOLDER("C:¥¥Archives¥¥2011¥¥February¥¥Doc.txt";*)
 // the "Doc.txt" file is ignored
```

# DELETE DOCUMENT

DELETE DOCUMENT ( document )

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name or Full document pathname |

## Description

The **DELETE DOCUMENT** command deletes the document whose name you pass in *document*.

If the document name or the entered path name is incorrect, an error is generated. This is also the case if you try to delete an open document.

**DELETE DOCUMENT** doesn't accept an empty string argument for *document*. If an empty string is used, the Open File dialog box is not displayed and an error is generated.

**WARNING: DELETE DOCUMENT** can delete any file on a disk. This includes documents created with other applications as well as the applications themselves. **DELETE DOCUMENT** should be used with extreme caution. Deleting a document is a permanent operation and cannot be undone.

## Example 1

The following example deletes the document named Note:

```
DELETE DOCUMENT("Note") ` Delete the document
```

## Example 2

See example for the **APPEND DATA TO PASTEBOARD** command.

## System variables and sets

Deleting a document sets the OK system variable to 1. If **DELETE DOCUMENT** can't delete the document, the OK system variable is set to 0.

# DELETE FOLDER

DELETE FOLDER ( folder {; deleteOption} )

| Parameter | Type | | Description |
|---|---|---|---|
| folder | String | ⇒ | Name or full path of the folder to be deleted |
| deleteOption | Longint | ⇒ | Folder deletion option |

## Description

The **DELETE FOLDER** command deletes the folder whose name or full path has been passed in *folder*.

By default, for security reasons, if you omit the *deleteOption* parameter, **DELETE FOLDER** only allows empty folders to be deleted. If you want the command to be able to delete non-empty folders, you must use the *deleteOption* parameter. In *deleteOption*, you can pass one of the following constants, found in the "**System Documents**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Delete only if empty | Longint | 0 | Deletes folder only when it is empty |
| Delete with contents | Longint | 1 | Deletes folder along with everything it contains |

- When Delete only if empty (0) is passed or if you omit the *deleteOption* parameter:
  - The folder specified in the *folder* parameter is only deleted if it is empty; otherwise, the command does nothing and an error -47 (The file is already open, or the folder is not empty) is generated.
  - If the folder specified does not exist , the error -120 (Tried to access a file by using a pathname that specifies a non existing directory) is generated.

- When Delete with contents (1) is passed:
  - The folder along with all of its contents are deleted.
    **Warning:** Even when this folder and/or its contents are locked or set to read-only, if the current user has suitable access rights, they are still deleted.
  - If this folder, or any of the files it contains, cannot be deleted, deletion is aborted as soon as the first inaccessible element is detected, and an error(*) is returned. In this case, the folder may be only partially deleted. When deletion is aborted, you can use **GET LAST ERROR STACK** command to retrieve the name and path of the offending file.
  - If the folder specified does not exist, the command does nothing and no error is returned.

    (*) under Windows: -54 (Attempt to open locked file for writing)
    under OS X: -45 (The file is locked or the pathname is not correct)

You can intercept these errors using a method installed by the **ON ERR CALL** command.

# Document creator

Document creator ( document ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name or Full document pathname |
| Function result | String | ⇐ | Empty string (Windows) or File Creator (Mac OS) |

## Description

The **Document creator** command returns the creator of the document whose name or pathname you pass in *document*.

On Windows, **Document creator** returns an empty string.

# DOCUMENT LIST

DOCUMENT LIST ( pathname ; documents {; options} )

| Parameter | Type | | Description |
|---|---|---|---|
| pathname | String | ⇒ | Pathname to volume, directory or folder |
| documents | Text array | ⇐ | Names of the documents present at this location |
| options | Longint | ⇒ | Options for building list |

## Description

The **DOCUMENT LIST** command populates the Text array *documents* with the names of the documents located at the location you pass in *pathname*.

**Note:** You must pass an absolute pathname in the *pathname* parameter.

By default, if you omit the *options* parameter, only the names of documents are returned in the *documents* array. You can modify this by passing, in the *options* parameter, one or more of the following constants, found in the **System Documents** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Absolute path | Longint | 2 | The *documents* array contains absolute pathnames |
| Ignore invisible | Longint | 8 | Invisible documents are not listed |
| Posix path | Longint | 4 | The *documents* array contains Posix format pathnames |
| Recursive parsing | Longint | 1 | The *documents* array contains all files and subfolders of the specified folder |

**Notes:**

- With the Recursive parsing option in relative mode (option 1 only), the paths of documents located in subfolders begin with the ":" or "¥" characters depending on the platform.
- With the Posix path option in relative mode (option 4 only), paths do not start with "/".
- With the Posix path option in absolute mode (option 4 + 2), paths always begin with "/".

If there are no documents at the specified location, the command returns an empty array. If the pathname you pass in *pathname* is invalid, **DOCUMENT LIST** generates a file manager error that you can intercept using an **ON ERR CALL** method.

## Example 1

List of all documents in a folder (default syntax):

```
DOCUMENT LIST("C:¥¥";arrFiles)
```

-> arrFiles:
Text1.txt
Text2.txt

## Example 2

List of all documents in a folder in absolute mode:

```
DOCUMENT LIST("C:¥¥";arrFiles; Absolute path)
```

-> arrFiles:
C:¥Text1.txt
C:¥Text2.txt

## Example 3

List of all documents in recursive (relative) mode:

```
DOCUMENT LIST("C:¥¥";arrFiles;Recursive parsing)
```

-> arrFiles:
Text1.txt
Text2.txt
¥Folder1¥Text3.txt
¥Folder1¥Text4.txt
¥Folder2¥Text5.txt
¥Folder2¥Folder3¥Picture1.png

## Example 4

List of all documents in recursive absolute mode:

```
DOCUMENT LIST("C:¥¥MyFolder¥¥";arrFiles;Recursive parsing+Absolute path)
```

-> arrFiles:
C:¥MyFolder¥MyText1.txt
C:¥MyFolder¥MyText2.txt
C:¥MyFolder¥Folder1¥MyText3.txt
C:¥MyFolder¥Folder1¥MyText4.txt
C:¥MyFolder¥Folder2¥MyText5.txt
C:¥MyFolder¥Folder2¥Folder3¥MyPicture1.png

## Example 5

List of all documents in recursive Posix (relative) mode:

```
DOCUMENT LIST("C:¥¥MyFolder¥¥";arrFiles;Recursive parsing+Posix path)
```

-> arrFiles:
MyText1.txt
MyText2.txt
Folder1/MyText3.txt
Folder1/MyText4.txt
Folder2/MyText5.txt
Folder2/Folder3/MyPicture1.png

# Document to text

Document to text ( fileName {; charSet {; breakMode}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| fileName | String | ➡ | Document name or Pathname to document |
| charSet | Text, Longint | ➡ | Name or Number of character set |
| breakMode | Longint | ➡ | Processing mode for line breaks |
| Function result | Text | ⬅ | Text from the document |

## Description

The **Document to text** command lets you retrieve the contents of a file directly on disk in a 4D text variable or text field.

In *fileName*, pass the name or pathname of the file to be read. The file must exist on the disk, otherwise an error is generated. You can pass:

- just the file name, for example "myFile.txt": in this case, the file must be located next to the structure file of the application.
- a pathname relative to the structure file of the application, for example "¥¥docs¥¥myFile.txt" under Windows or ":docs:myFile.txt" under OS X.
- an absolute pathname, for example "c:¥¥app¥¥docs¥¥myFile.txt" under Windows or "MacHD:docs:myFile.txt" under OS X.

In *charSet*, you pass the character set to be used for reading the contents. You can pass a string containing the standard set name (for example “ISO-8859-1” or “UTF-8”) or its MIBEnum ID (longint). For more information about the list of character sets supported by 4D, refer to the description of the **CONVERT FROM TEXT** command.

If the document contains a Byte Order Mark (BOM), 4D uses the character set that it has set instead of the one specified in *charSet* (this parameter is then ignored).

If the document does not contain a BOM and if the *charSet* parameter is omitted, by default 4D uses the following character sets:

- under Windows: ANSI
- under OS X: MacRoman

In *breakMode*, you can pass a longint indicating the processing to apply to end-of-line characters in the document. You can pass one of the following constants, found in the "**System Documents**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Document unchanged | Longint | 0 | No processing |
| Document with CR | Longint | 3 | Line breaks are converted to OS X format: CR (carriage return) |
| Document with CRLF | Longint | 2 | Line breaks are converted to Windows format: CRLF (carriage return + line feed) |
| Document with LF | Longint | 4 | Line breaks are converted to Unix format: LF (line feed) |
| Document with native format | Longint | 1 | (Default) Line breaks are converted to the native format of the operating system: CR (carriage return) under OS X, CRLF (carriage return + line feed) under Windows |

By default, when you omit the *breakMode* parameter, line breaks are processed in native mode (1).

**Note:** This command does not modify the OK variable. In case of failure, an error is generated that you can intercept using a method installed by the **ON ERR CALL** command.

## Example

Given the following text document (fields are separated by tabs):

```
id    name    price    vat
3    4D Tags    99    19.6
```

When you execute this code:

```
$Text:=Document to text("products.txt")
```

... you get:

```
// $Text = "id¥tname¥tprice¥tvat¥r¥n3¥t4D Tags¥t99 ¥t19.6"
// ¥t = tab
// ¥r = CR
```

# Document type

Document type ( document ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name |
| Function result | String | ⇐ | Windows file extension (1 to 3-character string) or Mac OS file type (4-character string) |

## Description

The **Document type** command returns the extension or the type of the document whose name or pathname you pass in *document*.

On Windows, **Document type** returns the file extension of the document (i.e. *'DOC'* for a Microsoft Word document, *'EXE'* for an executable file, and so on) or the corresponding Mac OS-based 4 characters file type if this latter has been mapped with its equivalent Windows file extension by 4D (i.e. *'TEXT'* for the *'TXT'* file extension) or by a prior call to **MAP FILE TYPES**.

On Macintosh, **Document type** returns, if specified, the 4-characters file type of the document (i.e. *'TEXT'* for a Text document, *'APPL'* for a double-clickable application and so on).

### Compatibility Note

The use of Mac OS file types is obsolete under Mac OS X. Like Windows, file identification is now based on the suffix of its name (see the **System Documents** section). For compatibility's sake, it nevertheless remains possible to read the Mac OS type of documents when it has been specified using **MAP FILE TYPES**.

# FOLDER LIST

FOLDER LIST ( pathname ; directories )

| Parameter | Type | | Description |
|---|---|---|---|
| pathname | String | ⇒ | Pathname to volume, directory or folder |
| directories | String array | ⇐ | Names of the directories present at this location |

## Description

The **FOLDER LIST** command populates the Text or String array *directories* with the names of the folders located at the pathname you pass in *pathname*.

**Note:** The *pathname* parameter only accepts absolute pathnames.

If there are no folders at the specified location, the command returns an empty array. If the pathname you pass in *pathname* is invalid, **FOLDER LIST** generate a file manager error that you can intercept using an **ON ERR CALL** method.

# GET DOCUMENT ICON

GET DOCUMENT ICON ( docPath ; icon {; size} )

| Parameter | Type | | Description |
|---|---|---|---|
| docPath | String | ➡ | Name or path of document to get icon, or Empty string for standard Open File dialog box |
| icon | Picture | ➡ | Picture variable or field |
| size | Longint | ➡ | Size of the returned picture (in pixels) |

## Description

The **GET DOCUMENT ICON** command returns, in the 4D picture variable or field *icon*, the icon of the document whose name or complete pathname is passed in *docPath*. *docPath* can specify a file of any type (executable, document, shortcut or alias, etc.) or a folder.

*docPath* contains the full pathname of the document. You can also pass the document name only or a relative pathname, in this case the document must be placed in the database current working directory (usually, the folder containing the database structure file).
If you pass an empty string in *docPath*, the standard Open File dialog box appears. The user can then select the file to read. Once the dialog box is validated, the Document system variable contains the full pathname to the selected file.

Pass a 4D picture field or variable in *icon*. After the command is executed, this parameter contains the icon of the file (PICT format).

The optional *size* parameter sets the dimensions in pixels of the returned icon. This value actually represents the side length of the square including the icon. Icons are usually defined in 32x32 pixels (“large icons”) or 16x16 pixels (“small icons”). If you pass 0 or omit this parameter, the largest available icon is returned.

# Get document position

Get document position ( docRef ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| docRef | DocRef | ⇒ | Document reference number |
| Function result | Real | ⇐ | File position (expressed in bytes) from the beginning of the file |

## Description

This command operates only on a document that is currently open whose document reference number you pass in *docRef*.

**Get document position** returns the position, starting from the beginning of the document, where the next read (**RECEIVE PACKET**) or write (**SEND PACKET**) will occur.

# GET DOCUMENT PROPERTIES

GET DOCUMENT PROPERTIES ( document ; locked ; invisible ; created on ; created at ; modified on ; modified at )

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name |
| locked | Boolean | ⇐ | Locked (True) or unlocked (False) |
| invisible | Boolean | ⇐ | Invisible (True) or visible (False) |
| created on | Date | ⇐ | Creation date |
| created at | Time | ⇐ | Creation time |
| modified on | Date | ⇐ | Last modification date |
| modified at | Time | ⇐ | Last modification time |

## Description

The **GET DOCUMENT PROPERTIES** command returns information about the document whose name or pathname you pass in *document*.

After the call:

- *locked* returns True if the document is locked. A locked document cannot be modified.
- *invisible* returns True if the document is hidden.
- *created on* and *created at* return the date and time when the document was created.
- *modified on* and *modified at* return the date and time when the document modified for the last time.

## Example

You have created a documentation database and you would like to export all the records you created in the database to documents on disk. Because the database is regularly updated you want to write an export algorithm that create or recreate each document on disk if the document does not exist or if the corresponding record has been modified after the document was saved for the last time. Consequently, you need to compare the date and time of modification of a document (if it exists) with its corresponding record. For illustrating this example, we use the table whose definition is shown below:


Rather than saving both a date and time values into each record, you can save a “time stamp” value which expresses the number of seconds elapsed between an arbitrary anterior date and time (in this example we use Jan, 1st 1995 at 00:00:00) and the date and time when the record was saved.

In our example, the field *[Documents]Creation Stamp* holds the time stamp when the record was first created and the field *[Documents]Modification Stamp* holds the time stamp when the record was last modified.

The ***Time stamp*** project method listed below calculates the time stamp for a specific date and time or for the current date and time if no parameters are passed:

```
[ ` Time stamp Project Method
  ` Time stamp { ( date ; Time ) } -> Long
  ` Time stamp { ( date ; Time ) } -> Number of seconds since Jan, 1st 1995

C_DATE($1;$vdDate)
C_TIME($2;$vhTime)
C_LONGINT($0)

If(Count parameters=0)
   $vdDate:=Current date
   $vhTime:=Current time
Else
```

```
   $vdDate:=$1
   $vhTime:=$2
End if
$0:=(($vdDate-!01/01/95!)*86400)+$vhTime
```

**Note:** Using this method, you can encode dates and times from the *01/01/95* at *00:00:00* to the *01/19/2063* at *03:14:07* which cover the long integer range *0* to *2^31* minus one.

Conversely, the ***Time stamp to date*** and ***Time stamp to time*** project methods listed below allow extracting the date and the time stored into a time stamp:

```
 ` Time stamp to date Project Method
 ` Time stamp to date ( Long ) -> Date
 ` Time stamp to date ( Time stamp ) -> Extracted date

C_DATE($0)
C_LONGINT($1)

$0:=!01/01/95!+($1¥86400)

 ` Time stamp to time Project Method
 ` Time stamp to time ( Long ) -> Date
 ` Time stamp to time ( Time stamp ) -> Extracted time

C_TIME($0)
C_LONGINT($1)

$0:=Time(Time string(†00:00:00†+($1%86400)))
```

For ensuring that the records have their time stamps correctly updated no matter the way they are created or modified, we just need to enforce that rule using the trigger of the table *[Documents]*:

```
 // Trigger for table [Documents]
Case of
   :(Trigger event=Save New Record Event)
      [Documents]Creation Stamp:=Time stamp
      [Documents]Modification Stamp:=Time stamp
   :(Trigger event=Save Existing Record Event)
      [Documents]Modification Stamp:=Time stamp
End case
```

Once this is implemented in the database, we have all we need to write the project method ***CREATE DOCUMENTATION*** listed below. We use of **GET DOCUMENT PROPERTIES** and **SET DOCUMENT PROPERTIES** for handling the date and time of creation and modification of the documents.

```
 ` CREATE DOCUMENTATION Project Method

C_STRING(255;$vsPath;$vsDocPathName;$vsDocName)
C_LONGINT($vlDoc)
C_BOOLEAN($vbOnWindows;$vbDoIt;$vbLocked;$vbInvisible)
C_TIME($vhDocRef;$vhCreatedAt;$vhModifiedAt)
C_DATE($vdCreatedOn;$vdModifiedOn)

If(Application type=4D Client)
 ` If we are running 4D Client, save the documents
 ` locally on the Client machine where 4D Client is located
   $vsPath:=Long name to path name(Application file)
Else
 ` Otherwise, save the documents where the data file is located
   $vsPath:=Long name to path name(Data file)
End if
 ` Save the documents in a directory we arbitrarily name "Documentation"
$vsPath:=$vsPath+"Documentation"+Char(Directory symbol)
 ` If this directory does not exist, create it
If(Test path name($vsPath)#Is a folder)
   CREATE FOLDER($vsPath)
End if
 ` Establish the list of the already existing documents
 ` because we'll have to delete the obsolete ones, in other words,
```

```
  ` the documents whose corresponding records have been deleted.
ARRAY STRING(255;$asDocument;0)
DOCUMENT LIST($vsPath;$asDocument)
  ` Select all the records from the [Documents] table
ALL RECORDS([Documents])
  ` For each record
$vlNbRecords:=Records in selection([Documents])
$vlNbDocs:=0
$vbOnWindows:=On Windows
For($vlDoc;1;$vlNbRecords)
  ` Assume we will have to (re)create the document on disk
   $vbDoIt:=True
  ` Calculate the name and the path name of the document
   $vsDocName:="DOC"+String([Documents]Number;"00000")
   $vsDocPathName:=$vsPath+$vsDocName
  ` Does this document already exist?
   If(Test path name($vsDocPathName+".HTM")=Is a document)
  ` If so, remove the document from the list of the documents
  ` that may end up deleted
      $vlElem:=Find in array($asDocument;$vsDocName+".HTM")
      If($vlElem>0)
         DELETE FROM ARRAY($asDocument;$vlElem)
      End if
  ` Was the document saved after the last time the record was modified?
      GET DOCUMENT PROPERTIES($vsDocPathName+".HTM";$vbLocked;$vbInvisible;$vdCreatedOn;$vhCreatedAt;
      $vdModifiedOn;$vhModifiedAt)
      If(Time stamp($vdModifiedOn;$vhModifiedAt)>=[Documents]Modification Stamp)
  ` If so, we do not need to recreate the document
         $vbDoIt:=False
      End if
   Else
  ` The document does not exist, reset these two variables so
  ` we know we'll have to compute them before setting the final properties
  ` of the document
      $vdModifiedOn:=!00/00/00!
      $vhModifiedAt:=?00:00:00?
   End if
  ` Do we need to (re)create the document?
   If($vbDoIt)
  ` If so, increment the number of updated documents
      $vlNbDocs:=$vlNbDocs+1
  ` Delete the document if it already exists
      DELETE DOCUMENT($vsDocPathName+".HTM")
  ` And create it again
      If($vbOnWindows)
         $vhDocRef:=Create document($vsDocPathName;"HTM")
      Else
         $vhDocRef:=Create document($vsDocPathName+".HTM")
      End if
      If(OK=1)
  ` Here write the contents of the document
         CLOSE DOCUMENT($vhDocRef)
         If($vdModifiedOn=!00/00/00!)
  ` The document did not exist, set the modification date and time
  ` to their right values
            $vdModifiedOn:=Current date
            $vhModifiedAt:=Current time
         End if
  ` Change the properties of the document so its date and time of creation
  ` are made equal to those of the corresponding record
         SET DOCUMENT PROPERTIES($vsDocPathName+".HTM";$vbLocked;$vbInvisible;
         Time stamp to date([Documents]Creation Stamp);
         Time stamp to time([Documents]Creation Stamp);
         $vdModifiedOn;$vhModifiedAt)
      End if
   End if
  ` Just to know what's going on
   SET WINDOW TITLE("Processing Document "+String($vlDoc)+" of "+String($vlNbRecords))
   NEXT RECORD([Documents])
End for
```

```
   ` Delete the obsolete documents, in other words
   ` those which are still in the array $asDocument
  For($vlDoc;1;Size of array($asDocument))
     DELETE DOCUMENT($vsPath+$asDocument{$vlDoc})
     SET WINDOW TITLE("Deleting obsolete document: "+Char(34)+$asDocument{$vlDoc}+Char(34))
  End for
   ` We're done
  ALERT("Number of documents processed: "+String($vlNbRecords)+Char(13)+"Number of documents updated:
 "+String($vlNbDocs)+Char(13)+"Number of documents deleted: "+String(Size of array($asDocument)))
```

# Get document size

Get document size ( document {; *} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| document | String, DocRef | ⇒ | Document reference number or Document name |
| * | Operator | ⇒ | On Mac OS only: - if omitted, size of data fork - if specified, size of resource fork |
| Function result | Real | ⇐ | Size (expressed in bytes) of the document |

## Description

The **Get document size** command returns the size, expressed in bytes, of a document.

If the document is open, you pass its document reference number in *document*.

If the document is not open, you pass its name or pathname in *document*.

On Macintosh, if you do not pass the optional * parameter, the size of the data fork is returned. If you do pass the * parameter, the size of the resource fork is returned.

# Get localized document path

Get localized document path ( relativePath ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| relativePath | Text | ⇒ | Relative pathname of document for which we want to obtain localized version |
| Function result | Text | ⇐ | Absolute pathname of localized document |

## Description

The **Get localized document path** command returns the complete (absolute) pathname of a document designated by *relativePath* and located in a xxx.lproj folder.

This command must be used within a multi-language application architecture based on the presence of a **Resources** folder and *xxx.lproj* subfolders (where xxx represents a language). With this architecture, 4D automatically supports localized files of the .xliff type as well as pictures, but you may need to use the same mechanism for other types of files.

Pass the relative pathname of the document to be searched for in *relativePath*. The path entered must be relative to the first level of the "xxx.lproj" folder of the database. The command will return a complete pathname using the "xxx.lproj" folder corresponding to the current language of the database.

**Note:** The current language is either set automatically by 4D according to the contents of the **Resources** folder (see the **Get database localization** command), or via the **SET DATABASE LOCALIZATION** command).

You can express the contents of the *relativePath* parameter using a system or a POSIX syntax. For example:

- xsl/log.xsl (POSIX syntax: can be used under Mac OS or Windows)
- xsllog.xsl (Windows only)
- xsl:log.xsl (Mac OS only)

The absolute pathname returned by the command is always expressed in the system syntax.

**4D Server:** In remote mode, the command returns the path of the **Resources** folder on the client machine if the command is called from a client process.

4D looks for the file while respecting a sequence that allows all the cases of multi-language applications to be processed. At each step, 4D checks for the presence of *relativePath* in the folder corresponding to the language and returns the complete path when it succeeds. If *relativePath* is not found or if the folder does not exist, 4D passes to the next step. Here are the folders for each of the different search stages:

*Current language (e.g.: fr-ca)*
*Current language without region (e.g.: fr)*
*Language loaded by default on startup (e.g.: es-ga)*
*Language loaded by default on startup without region (e.g.: es)*
*First .lproj folder found (e.g.: en.lproj)*
*First level of Resources folder*

If *relativePath* is not found in any of these locations, the command returns an empty string.

## Example

For the purpose of transforming an XML or HTML file, you want to use a "log.xsl" transformation file. This file differs depending on the current language. You therefore want to know which "log.xsl" file path to use.
Here are the contents of the Resources folder:


To use a .xsl file adapted to the current language, you simply need to pass:

```
$myxsl:=Get localized document path("xsl/log.xsl")
```

If the current language is, for example, French Canadian (fr-ca), the command returns:

- under Windows: C:¥users¥…¥…¥…¥resources¥fr_ca.lproj¥xsl¥log.xsl
- under Mac OS: "HardDisk:users:…:…:…:resources:fr_ca.lproj:xsl:log.xsl"

# MAP FILE TYPES

MAP FILE TYPES ( macOS ; windows ; context )

| Parameter | Type | | Description |
|---|---|---|---|
| macOS | String | ⇒ | Mac OS file type (4-character string) |
| windows | String | ⇒ | Windows file extension |
| context | String | ⇒ | String displayed in List of Types drop-down list of the Windows file dialog boxes |

## Description

**MAP FILE TYPES** lets you associate a Windows file extension with a Macintosh file type.

You need to call this routine only once to establish a mapping for an entire worksession with a database. Once the call has been made, the commands **Create document**, **Append document**, and **Open resource file** while running on Windows will automatically substitute the Windows file extension for the Macintosh file type you actually pass as a parameter to the routine.

In the *macOS* parameter you pass a 4-character Macintosh file type. If you do not pass a 4-character string, the command does nothing and generates an error.

In the *windows* parameter you pass a 1- to X-character Windows file extension. If you do not pass a 1 to 3-character string, the command does nothing and generates an error.

In the *context* parameter you pass the string that will be displayed in the List Files of Type drop-down list of the Windows Open File dialog box. The context string is limited to 32 characters; additional characters are ignored.

**IMPORTANT:** Once you have mapped a Windows file extension to a Macintosh file type, you cannot change or delete this mapping within a single work session. If you need to change a mapping while developing and debugging a 4D application, reopen the database and remap the file extension.

## Example

The following line of 4D code (that could be part of the **Startup** database method) maps the Macintosh MS-Word file type *"WDBN"* to the Windows file extension *".DOC"*:

```
MAP FILE TYPES("WDBN";"DOC";"Word documents")
```

Once the call above has been made, the following code will display only Word documents in the Open file dialog on Windows and Macintosh:

```
$DocRef:=Open document("";"WDBN")
If(OK=1)
   ...
End if
```

# MOVE DOCUMENT

MOVE DOCUMENT ( srcPathname ; dstPathname )

| Parameter | Type | | Description |
|---|---|---|---|
| srcPathname | String | ➡ | Full pathname to existing document |
| dstPathname | String | ➡ | Destination pathname |

## Description

The **MOVE DOCUMENT** command moves or renames a document.

You specify the full pathname to the document in *srcPathname* and the new name and/or new location for the document in *dstPathname*.

**Warning:** Using **MOVE DOCUMENT**, you can move a document from and to any directory on the same volume. If you want to move a document between two distinct volumes, use **COPY DOCUMENT** to "move" the document then delete the original copy of the document using **DELETE DOCUMENT**.

## Example 1

The following example renames the document *DocName*:

```
MOVE DOCUMENT("C:¥¥FOLDER¥¥DocName";"C:¥¥FOLDER¥¥NewDocName")
```

## Example 2

The following example moves and renames the document *DocName*:

```
MOVE DOCUMENT("C:¥¥FOLDER1¥¥DocName";"C:¥¥FOLDER2¥¥NewDocName")
```

## Example 3

The following example moves the document *DocName*:

```
MOVE DOCUMENT("C:¥¥FOLDER1¥¥DocName";"C:¥¥FOLDER2¥¥DocName")
```

**Note:** In the last two examples, the destination folder *"C:¥¥FOLDER2"* must exist. The **MOVE DOCUMENT** command only moves a document; it does not create folders.

# Open document

Open document ( document {; fileType}{; mode} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name or Full document pathname or Empty string for standard file dialog box |
| fileType | String | ⇒ | List of types of documents to be screened, or "*" to not screen the documents |
| mode | Longint | ⇒ | Document's opening mode |
| Function result | DocRef | ⇐ | Document reference number |

## Description

The **Open document** command opens the document whose name or pathname you pass in *document*.

If you pass an empty string in *document*, the Open File dialog box is presented, and you then select the document to be open. If you cancel the dialog, no document is opened; **Open document** returns a null DocRef and sets the OK variable to 0.

- If the document is correctly opened, **Open document** returns its document reference number and sets the OK variable to 1.
- If the document is already open in Read mode and the *mode* parameter is omitted, **Open document** opens the document in Read/Write mode by default and sets the OK variable to 1.
- If the document is already open in Read/Write mode and you try to open it in Write mode, an error (-43) is generated. However, you can open it in Read only mode, then the OK variable is set to 1.
- If the document does not exist, an error is generated.

In the *fileType* parameter, pass the type(s) of file(s) that can be selected in the opening dialog box. You can pass a list of several types separated by a ; (semi-colon). For each type set, an item will be added to the menu used for choosing the type in the dialog box.

Under Mac OS, you can pass either a standard Mac OS type (TEXT, APPL, etc.), or a UTI (Uniform Type Identifier) type. UTIs are defined by Apple in order to meet standardization needs for file types. For example, "public.text" is the UTI type of text type files. For more information about UTIs, refer to the following address: https://developer.apple.com/library/mac/#documentation/FileManagement/Conceptual/understanding_utis/understand_utis_intro/understand_utis_intro.html.

Under Windows, you can also pass a standard Mac OS file type — 4D makes the correspondence internally — or file extensions (.txt, .exe, etc.). Note that under Windows, the user can "force" the display of all file types by entering *.* in the dialog box. However, in this case, 4D will carry out an additional check of the selected file types: if the user selects an unauthorized file type, the command returns an error.

If you do not want to restrict the displayed files to one or more types, pass the "*" (star) string or ".*" in *fileType*.

The optional *mode* parameter allows you to define how *document* is to be opened. Four different open file modes are possible. 4D offers the following predefined constants, located in the "**System Documents**" theme:

| Constant | Type | Value |
|---|---|---|
| Get Pathname | Longint | 3 |
| Read and Write | Longint | 0 |
| Read Mode | Longint | 2 |
| Write Mode | Longint | 1 |

If a document is open, **Open document** initially sets the file position at the beginning of the document while **Append document** sets it at the end of the document.

Once you have opened a document, you can read and write in the document using the **RECEIVE PACKET** and **SEND PACKET** commands that you can combine with the **Get document position** and **SET DOCUMENT POSITION** commands in order to directly access any part of the document.

Do not forget to eventually call **CLOSE DOCUMENT** for the document.

## Example 1

The following example opens an existing document called Note, writes the string "Good-bye" into it, and closes the document. If the document already contains the string "Hello", this string would be overwritten:

```
C_TIME(vhDoc)
vhDoc:=Open document("Note.txt";Read and Write) ` Open a document called Note
If(OK=1)
   SEND PACKET(vhDoc;"Good-bye") ` Write one word into the document
   CLOSE DOCUMENT(vhDoc) ` Close the document
End if
```

## Example 2

You can read a document even if it is already open in write mode:

```
vDoc:=Open document("PassFile";"TEXT") ` The file is open
 ` Before the file is closed, it is possible to consult it in read-only mode:
vRef:=Open document("PassFile";"TEXT";Read Mode)
```

## System variables and sets

If the document is correctly opened, the OK system variable is set to 1; otherwise, it is set to 0. After the call, the Document system variable contains the full name of the document.
If you call **Open document** with a mode of 3, the function returns ?00:00:00? (no document reference). The document is not opened but the Document and OK system variables are updated:

- OK is equal to 1.
- Document contains the full pathname and the name of *document*.

**Note:** If you pass an empty string in *document*, an open file dialog box appears. If the user chooses a document and clicks the OK button, *document* is set to the path of the document the user selected and OK is set to 1. If the user clicked the Cancel button, OK is set to 0.

# RESOLVE ALIAS

RESOLVE ALIAS ( aliasPath ; targetPath )

| Parameter | Type | | Description |
|---|---|---|---|
| aliasPath | String | ⇒ | Name or access path of the alias/shortcut |
| targetPath | String | ⇐ | Name or access path of the alias/shortcut target |

## Description

The **RESOLVE ALIAS** command returns the full path to the target file or folder of the alias (named shortcut under Windows).

The full path to the alias is passed in *aliasPath*.

Once the command has been executed, the *targetPath* variable contains the full path to the target file or folder of the alias and the OK system variable is set to 1.

If the path passed in *aliasPath* corresponds to a file and not an alias, *targetPath* returns the path of the file and the OK system variable is set to 0.

## System variables and sets

If *aliasPath* does specify an alias/shortcut, the OK system variable is set to 1. If *aliasPath* specifies a standard file, the OK system variable is set to 0.

# Select document

Select document ( directory ; fileTypes ; title ; options {; selected} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| directory | Text, Longint | ⇒ | • Directory access path to display by default in the document selection dialog box, or<br>• Empty string to display default user folder (“My documents” under Windows, “Documents” under Mac OS), or<br>• Number of the memorized access path |
| fileTypes | Text | ⇒ | List of types of documents to filter, or "*" to not filter documents |
| title | Text | ⇒ | Title of the selection dialog box |
| options | Longint | ⇒ | Selection option(s) |
| selected | Text array | ⇐ | Array containing the list of access paths + names of selected files |
| Function result | String | ⇐ | Name of selected file (first file of the list in case of multiple selection) |

## Description

The **Select document** command displays a standard open document dialog box which allows the user to set one or more files and returns the name and/or full access path of the selected file(s).

The *directory* parameter indicates the folder whose contents are initially displayed in the open document dialog box. You can pass three types of values:

- a text containing the full access path of the folder to display.
- an empty string ("") to display the default user folder for the current opeerating system (“My documents” under Windows, “Documents” under Mac OS).
- a number of the memorized access path (from 1 to 32000) to display the associated folder.
  As such, you can store in memory the access path of the folder opened when the user clicked the selection button, in other words, the folder selected by the user. During the first call of an arbitrary number (for example, 5) the command displays the default user folder of the operating system (equivalent of passing an empty string). The user could also browse folders on the hard disk. When the user clicks on the selection button, the access path is memorized and associated with number 5. During future calls to number 5, the memorized access path will be used by default. If a new location is selected, path number 5 is updated.
  This mechanism lets you memorize up to 32,000 access paths. Under Windows, each path is kept for the session only. Under Mac OS, the paths are kept by the system and remain stored from one session to the next.

**Note:** This mechanism is the same as the one used by the **Select folder** command. The numbers of the memorized pathnames are shared by both commands.

Pass the type(s) of file(s) that can be selected in the open file dialog box in the *fileTypes* parameter. You can pass a list of several types separated by a ; (semi-colon). For each type defined, a row will be added in the type choice menu of the dialog box.

- Under Mac OS, you can pass either a standard Mac OS type (TEXT, APPL, etc.), or a UTI (Uniform Type Identifier) type. UTI types have been defined by Apple in order to meet requirements concerning the standardization of file types. For example, "public.text" is the UTI type of text type files. For more information about UTIs, please refer to the following address: Uniform Type Identifiers Overview on developer.apple.com.
- Under Windows, you can also pass a standard Mac OS type file — 4D performs the conversion internally — or the file extensions (.txt, .exe, etc.). Please note that under Windows, the user can “force” the display of all document types by entering *.* in the dialog box. However, in this case, 4D will perform an additional verification of the types of files selected: if the user selects an unauthorized file type, the command returns an error.

If you do not want to restrict the files displayed to one or more types, pass the "*" (star) or ".*" string in *fileTypes*.

Pass the label that must appear in the dialog box in the *title* parameter. By default, if you pass an empty string, the label “Open” is displayed.

The *options* parameter allows you to specify advanced functions that are allowed in an open file dialog box. 4D provides the following pre-defined constants in the **System Documents** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Alias selection | Longint | 8 | Authorizes the selection of shortcuts (Windows) or aliases (Mac OS) as document. By default, if this constant is not used, when an alias or shortcut is selected, the command will return the access path of the targeted element. When you pass the constant, the command returns the path of the alias or shortcut itself. |
| File name entry | Longint | 32 | Allows user to enter a file name in a 'Save as' dialog box. No file is saved and it is up to the developer to create a file in response to this action (the Document system variable is updated). In this context, the *directory* parameter may contain the path to a file instead of a directory. The file name will be used as the suggested file name in the Save as text field. The parent directory will be used as default path. |
| Multiple files | Longint | 1 | Authorizes the simultaneous selection of several files using the key combinations **Shift+click** (adjacent selection) and **Ctrl+click** (Windows) or **Command+click** (Mac OS). In this case, the *selected* parameter, if passed, contains the list of all selected files. By default, if this constant is not used, the command will not allow the selection of multiple files. |
| Package open | Longint | 2 | (Mac OS only): Authorizes the opening of packages as folders and thus the viewing /selection of their contents. By default, if this constant is not used, the command will not allow the opening of packages. |
| Package selection | Longint | 4 | (Mac OS only): Authorizes the selection of packages as entities. By default, if this constant is not used, the command will not allow the selection of software packages as such. |
| Use sheet window | Longint | 16 | (Mac OS only): Displays the selection dialog box in the form of a sheet window (this option is ignored under Windows). Sheet windows are specific to the Mac OS X interface which have graphic animation (for more information, refer to the **Window Types** section). By default, if this constant is not used, the command will display a standard dialog box. |

If you do not want to use an option, pass 0 in the *options* parameter.

The optional *selected* parameter allows you to get the full access path (access path + name) of every file selected by the user. The command creates, sizes and fills the array according to the user selection. This parameter is useful when the Multiple files option is used or when you want to find out the access path of the selected file (simply take the name of the file returned by the command from the value of the array). If no file is selected, the array is returned empty.

**Note:** Under Mac OS, a selected package is considered as a folder. The pathname returned in the *selected* array includes a final ":" character. For example: **Disk:Applications:4D:4D v11.4:US:4D Volume Desktop.app:**

The command returns the name (name + extension under Windows) of the selected file. If several files are selected, the command returns the name of the first file in the list of selected files. The list of files can be obtained in the *selected* parameter. If no file is selected, the command returns an empty string.

## Example 1

This example is used to specify a 4D data file:

```
C_LONGINT($platform)
PLATFORM PROPERTIES($platform)
If($platform=Windows)
   $DocType:=".4DD"
Else
   $DocType:="com.4d.4d.data-file" `UTI type
End if
$Options:=Alias selection+Package open+Use sheet window
$Doc:=Select document("";$DocType;"Select the data file";$Options)
```

## Example 2

Creation of a custom document by user:

```
$doc:=Select document(System folder(Documents folder)+"Report.pdf";"pdf";"Report name:";File name entry)
If(OK=1)
   BLOB TO DOCUMENT(Document;$blob) // $blob contains document to record
End if
```

## System variables and sets

If the command has been correctly executed and a valid document was selected, the system variable OK is set to 1 and the system variable Document will contain the full access path of the selected file.
If no file was selected (for example, if the user clicked on the **Cancel** button in the open file dialog box), the system variable OK is set to 0 and the system variable Document will be empty.

# Select folder

Select folder ( {message }{;}{ defaultPath {; options}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| message | String | ⇒ | Title of the window |
| defaultPath | String, Longint | ⇒ | • Default pathname or • Empty string to display the default user folder (“My documents” under Windows, “Documents” under Mac OS), or • Number of memorized pathname |
| options | Longint | ⇒ | Selection option(s)under Mac OS |
| Function result | String | ⇐ | Access path to the selected folder |

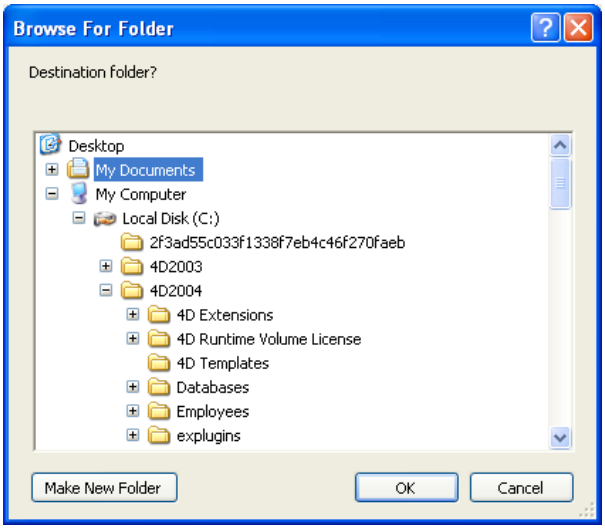
## Description

The **Select folder** command displays a dialog box that allows you to manually select a folder and then retrieve the complete access path to that folder. The optional *defaultPath* parameter can be used to designate the location of a folder that will be initially displayed in the folder selection dialog box.

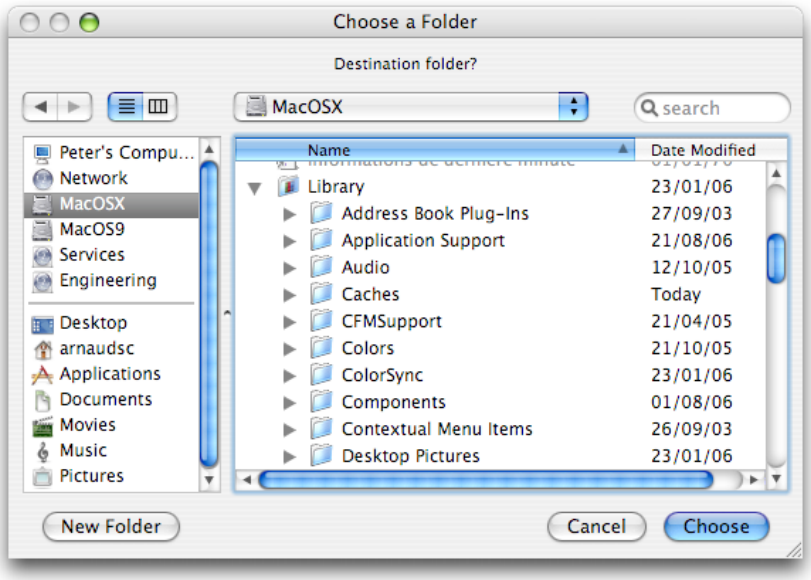
**Note:** This command does not modify 4D’s current folder.

The **Select folder** command displays a standard dialog box to browse through the workstation’s volumes and folders. The optional parameter *message* allows you to display a message in the dialog box. In the following examples, the message is "Destination folder?":

**Windows**




**Mac OS**



You can use the *defaultPath* parameter to provide a default folder location in the folder selection dialog box. You can pass three types of values in this parameter:

- The pathname of a valid folder using the syntax of the current platform.
- An empty string ("") to display the default user folder of the system ("My documents" under Windows, "Documents" under Mac OS).
- The number of a memorized pathname (from 1 to 32,000) to display the associated folder. This means that you can store in memory the pathname of the folder that is open when the user clicks on the selection button; in other words, the folder chosen by the user. When calling a random number (for instance, 5) the command displays the default user folder of the system (equivalent to passing an empty string). The user may then browse among the folders on their harddisk. When the user clicks on the selection button, the pathname is memorized and associated with the number 5. When the number 5 is called subsequently, the memorized pathnname will be used by default. If a new location is selected, the path number 5 will be updated, and so on.
  This mechanism can be used to memorize up to 32,000 pathnames. Under Windows, each path is only kept during the session. Under Mac OS, the paths remain memorized from one session to the next. If the pathname is incorrect, the *defaultPath* parameter is ignored.

**Note:** This mechanism is identical to the one used by the **Select document** command. The numbers of memorized pathnames are shared between both these commands.

The *options* parameter lets you benefit from additional functions under Mac OS. In this parameter, you can pass one of the following constants, found in the **System Documents** theme:

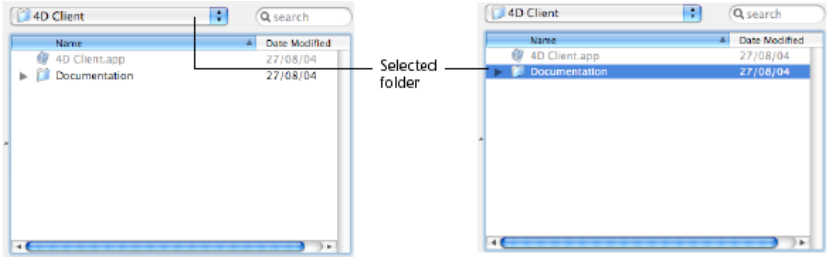
| Constant | Type | Value | Comment |
|---|---|---|---|
| Package open | Longint | 2 | (Mac OS only): Authorizes the opening of packages as folders and thus the viewing /selection of their contents. By default, if this constant is not used, the command will not allow the opening of packages. |
| Use sheet window | Longint | 16 | (Mac OS only): Displays the selection dialog box in the form of a sheet window (this option is ignored under Windows). Sheet windows are specific to the Mac OS X interface which have graphic animation (for more information, refer to the **DISPLAY SELECTION** section). By default, if this constant is not used, the command will display a standard dialog box. |

You can pass a single constant or a combination of both. These options are only taken into account under Mac OS. Under Windows, the *options* parameter is ignored if it is passed.

The user selects a folder and then clicks the **OK** button (on Windows) or the **Select** button (on Mac OS). The access path to the folder is then returned by the function.

- On Windows, the access path is returned in the following format:
  "C:¥Folder1¥Folder2¥SelectedFolder¥"
- On Mac OS, the access path is returned in the following format:
  "Hard Disk:Folder1:Folder2:SelectedFolder:"

**Note:** On Mac OS, depending on whether or not the name of the folder is selected in the dialog box, the access path that is returned to you may be different.




**4D Server:** This function allows you to view the volumes connected to the client workstations. It is not possible to call this function from a stored procedure.

If the user validates the dialog box, the **OK** system variable is set to 1. If the user clicks the **Cancel** button, the **OK** system variable is set to 0 and the function returns an empty string.

**Note:** On Windows, if the user selected some incorrect elements, such as "Workstation", "Trash can", and so on, the **OK** system variable is set to 0, even if the user validates the dialog box.

## Example

The following example allows you to select the folder in which the pictures in the picture library will be stored:

```
$PictFolder:=Select folder("Select a folder for your pictures.")
PICTURE LIBRARY LIST(pictRefs;pictNames)
For($n;1;Size of array(pictNames))
   GET PICTURE FROM LIBRARY(pictRefs{$n};$vStoredPict)
   WRITE PICTURE FILE($PictFolder+pictNames{$n};$vStoredPict)
End for
```

# SET DOCUMENT CREATOR

SET DOCUMENT CREATOR ( document ; fileCreator )

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name or Full document pathname |
| fileCreator | String | ⇒ | Mac OS file creator (4-character string) or empty string (Windows) |

## Description

The **SET DOCUMENT CREATOR** command sets the creator of the document whose name or pathname you pass in *document*.

You pass the new creator of the document in *fileCreator*.

This command does nothing on Windows.

See discussion about file creators in **System Documents**.

# SET DOCUMENT POSITION

SET DOCUMENT POSITION ( docRef ; offset {; anchor} )

| Parameter | Type | | Description |
|---|---|---|---|
| docRef | DocRef | ⇒ | Document reference number |
| offset | Real | ⇒ | File position (expressed in bytes) |
| anchor | Longint | ⇒ | 1 = In relation to the beginning of the file 2 = In relation to the end of the file 3 = In relation to current position |

## Description

This command operates only on a document currently open whose document reference number you pass in *docRef*.

**SET DOCUMENT POSITION** sets the position you pass in *offset* where the next read (**RECEIVE PACKET**) or write (**SEND PACKET**) will occur.

If you omit the optional *anchor* parameter, the position is relative to the beginning of the document. If you do specify the *anchor* parameter, you pass one of the values listed above.

Depending on the anchor you can pass positive or negative values in *offset*.

# SET DOCUMENT PROPERTIES

SET DOCUMENT PROPERTIES ( document ; locked ; invisible ; created on ; created at ; modified on ; modified at )

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name or Full document pathname |
| locked | Boolean | ⇒ | Locked (True) or Unlocked (False) |
| invisible | Boolean | ⇒ | Invisible (True) or Visible (False) |
| created on | Date | ⇒ | Creation date |
| created at | Time | ⇒ | Creation time |
| modified on | Date | ⇒ | Last modification date |
| modified at | Time | ⇒ | Last modification time |

## Description

The **SET DOCUMENT PROPERTIES** command changes the information about the document whose name or pathname you pass in *document*.

Before the call:

- Pass True in *locked* to lock the document. A locked document cannot be modified. Pass False in *locked* to unlock a document.
- Pass True in *invisible* to hide the document. Pass False in *invisible* to make the document visible in the desktop windows.
- Pass the document creation date and time in *created on* and *created at*.
- Pass the document last modification date and time in *modified on* and *modified at*.

The dates and times of creation and last modification are managed by the file manager of your system each time you create or access a document. Using this command, you can change those properties for special purpose. See example for the command **GET DOCUMENT PROPERTIES**.

# SET DOCUMENT SIZE

SET DOCUMENT SIZE ( docRef ; size )

| Parameter | Type | | Description |
|---|---|---|---|
| docRef | DocRef | ⇒ | Document reference number |
| size | Real | ⇒ | New size expressed in bytes |

## Description

The **SET DOCUMENT SIZE** command sets the size of a document to the number of bytes you pass in *size*.

If the document is open, you pass its document reference number in *docRef*.

On Macintosh, the size of the document's data fork is changed.

# SET DOCUMENT TYPE

SET DOCUMENT TYPE ( document ; fileType )

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document name or full document pathname |
| fileType | String | ⇒ | Windows file extension or Mac OS file type (4-character string) |

## Description

The **SET DOCUMENT TYPE** command sets the type of the document whose name or pathname you pass in *document*.

You pass the new type of the document in *fileType*.

See the discussion of file types in **System Documents** and **Document type**.

On Windows, this command modifies the file extension and therefore the value of *document*. For example, the instruction:

```
SET DOCUMENT TYPE("C:¥¥Docs¥¥Invoice.asc";"TEXT")
```

renames the file "Invoice.asc" to "Invoice.txt". In 4D, the Macintosh "TEXT" type corresponds to the Windows "txt" type.

If the type has no equivalent provided by 4D, you will have to pass the extension. For example, the following instruction renames the file "Invoice.asc" to "Invoice.zip":

```
SET DOCUMENT TYPE("C:¥¥Docs¥¥Invoice.asc";"zip")
```

# SHOW ON DISK

SHOW ON DISK ( pathname {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| pathname | String | ⇒ | Pathname of item to show |
| * | | ⇒ | If the item is a folder, show its contents |

## Description

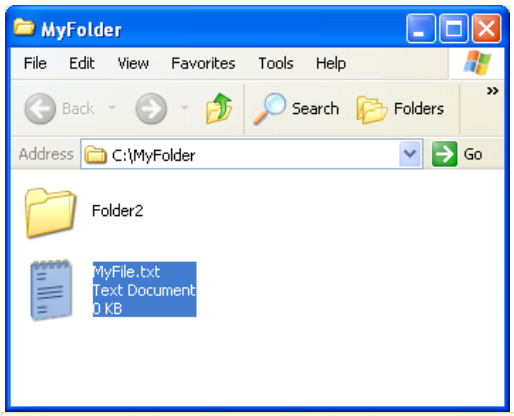
The **SHOW ON DISK** command displays the file or folder whose pathname was passed in the *pathname* parameter in a standard window of the operating system.
In a user interface, this command lets you designate the location of a specific file or folder.

By default, if *pathname* designates a folder, the command displays the level of the folder itself. If you pass the optional * parameter, the command opens the folder and displays its contents in the window. If *pathname* designates a file, the * parameter is ignored.

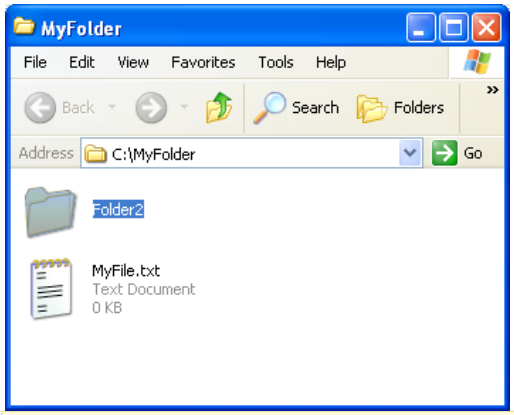
## Example

The following examples illustrate the operation of this command:

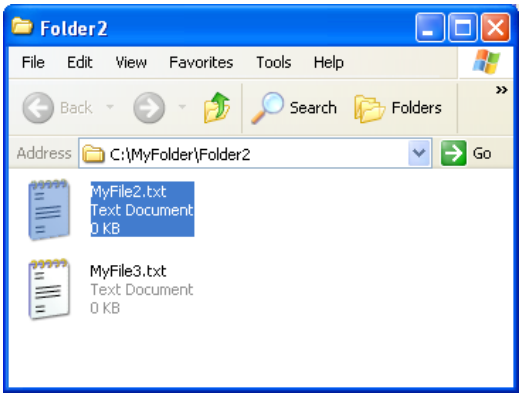
```
SHOW ON DISK("c:¥¥MyFolder¥¥MyFile.txt") ` Displays the designated file
```




```
SHOW ON DISK("c:¥¥MyFolder¥¥Folder2") ` Displays the designated folder
```




```
SHOW ON DISK("c:¥¥MyFolder¥¥Folder2";*) ` Displays the contents of the designated folder
```




## System variables and sets

The system variable OK is set to 1 if the command is executed correctly.

# Test path name

Test path name ( pathname ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| pathname | String | ⇒ | Pathname to directory, folder or document |
| Function result | Longint | ⇐ | 1, pathname refers to an existing document 0, pathname refers to an existing directory or folder <0, invalid pathname, OS file manager error code |

## Description

The **Test path name** function checks if a document or folder whose name or pathname you pass in *pathname* is present on the disk. You can pass either a relative or absolute pathname, expressed in the syntax of the current system.

If a document is found, **Test path name** returns 1. If a folder found, **Test path name** returns 0.

The following predefined constants are provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Is a document | Longint | 1 |
| Is a folder | Longint | 0 |

If no document nor folder is found, **Test path name** returns a negative value (i.e. -43 for File not found).

## Example

The following tests if the document “Journal” is present in the folder of the database, then creates it if it was not found:

```
If(Test path name("Journal")#Is a document)
   $vhDocRef:=Create document("Journal")
   If(OK=1)
      CLOSE DOCUMENT($vhDocRef)
   End if
End if
```

# TEXT TO DOCUMENT

TEXT TO DOCUMENT ( fileName ; text {; charSet {; breakMode}} )

| Parameter | Type | | Description |
|---|---|---|---|
| fileName | String | ➡ | Document name or Pathname to document |
| text | Text | ➡ | Text to store in the document |
| charSet | Text, Longint | ➡ | Name or Number of character set |
| breakMode | Longint | ➡ | Processing mode for line breaks |

## Description

The **TEXT TO DOCUMENT** command lets you write the *text* directly to a disk file.

In *fileName*, pass the name or pathname of the file to be written. If this file does not exist, it is created. When this file already exists on the disk, its prior contents are erased, except if it is already open, in which case, its contents are locked and an error is generated. In *fileName*, you can pass:

- just the file name, for example "myFile.txt": in this case, the file is placed next to the structure file of the application.
- a pathname relative to the structure file of the application, for example "¥¥docs¥¥myFile.txt" under Windows or ":docs:myFile.txt" under OS X.
- an absolute pathname, for example "c:¥¥app¥¥docs¥¥myFile.txt" under Windows or "MacHD:docs:myFile.txt" under OS X..

If you want the user to be able to indicate the name or location of the document, use the **Open document** or **Create document** commands, as well as the **Document** system variable.

**Note:** By default, documents generated by this command do not have an extension. You must pass an extension in *fileName*. You can also use the **SET DOCUMENT TYPE** command.

In *text*, pass the text to write to the disk. It can be a literal constant ("my text"), or a 4D text field or variable.

In *charSet*, you pass the character set to be used to write the document. You can pass a string containing the standard set name (for example “ISO-8859-1” or “UTF-8”) or its MIBEnum ID (longint). For more information about the list of character sets supported by 4D, refer to the description of the **CONVERT FROM TEXT** command. If a Byte Order Mark (BOM) exists for the character set, 4D inserts it into the document. If you do not specify a character set, by default 4D uses the "UTF_8" character set and a BOM.

In *breakMode*, you can pass a longint indicating the processing to apply to end-of-line characters before saving them in the file. You can pass one of the following constants, found in the "**System Documents**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Document unchanged | Longint | 0 | No processing |
| Document with CR | Longint | 3 | Line breaks are converted to OS X format: CR (carriage return) |
| Document with CRLF | Longint | 2 | Line breaks are converted to Windows format: CRLF (carriage return + line feed) |
| Document with LF | Longint | 4 | Line breaks are converted to Unix format: LF (line feed) |
| Document with native format | Longint | 1 | (Default) Line breaks are converted to the native format of the operating system: CR (carriage return) under OS X, CRLF (carriage return + line feed) under Windows |

By default, when you omit the *breakMode* parameter, line breaks are processed in native mode (1).

**Note:** This command does not modify the OK variable. In case of failure, an error is generated that you can intercept using a method installed by the **ON ERR CALL** command.

## Example 1

Here are some typical examples of using this command:

```
TEXT TO DOCUMENT("myTest.txt";"This is a test")
TEXT TO DOCUMENT("myTest.xml";"This is a test")
```

## Example 2

Example allowing the user to indicate the location of the file to create:

```
$MyTextVar:="This is a test"
ON ERR CALL("IO ERROR HANDLER")
$vhDocRef :=Create document("")
 // Store document with the ".txt" extension
 // In this case, the .txt extension is always added to the name; it is not possible to change it
If(OK=1) // If document has been created successfully
   CLOSE DOCUMENT($vhDocRef) //Closes the document
   TEXT TO DOCUMENT(Document;$MyTextVar )
 // We write the document
Else
 // Error management
End if
```

# VOLUME ATTRIBUTES

VOLUME ATTRIBUTES ( volume ; size ; used ; free )

| Parameter | Type | | Description |
|---|---|---|---|
| volume | String | ⇒ | Volume name |
| size | Real | ⇐ | Volume size expressed in bytes |
| used | Real | ⇐ | Used space expressed in bytes |
| free | Real | ⇐ | Free space expressed in bytes |

## Description

The **VOLUME ATTRIBUTES** command returns, expressed in bytes, the size, the used space and the free space for the volume whose name you pass in *volume*.

**Note:** If *volume* indicates a non-mounted remote volume, the OK variable is set to 0 and the three parameters return -1.

## Example

Your application includes some batch operations running the night or the week-end that store huge temporary files on disk. To make this process as automatic and flexible as possible, you write a routine that will automatically find the first volume whose free space is sufficient for your temporary files. You might write the following project method:

```
  ` Find volume for space Project Method
  ` Find volume for space ( Real ) -> String
  ` Find volume for space ( Space needed in bytes ) -> Volume name or Empty string

C_STRING(31;$0)
C_STRING(255;$vsDocName)
C_LONGINT($vlNbVolumes;$vlVolume)
C_REAL($1;$vlSize;$vlUsed;$vlFree)
C_TIME($vhDocRef)

  ` Initialize function result
$0:=""
  ` Protect all I/O operations with an error interruption method
ON ERR CALL("ERROR METHOD")
  ` Get the list of the volumes
ARRAY STRING(31;$asVolumes;0)
gError:=0
VOLUME LIST($asVolumes)
If(gError=0)
  ` If running on windows, skip the (usual) two floppy drives
   If(On Windows)
      $vlVolume:=Find in array($asVolumes;"A:¥¥")
      If($vlVolume>0)
         DELETE FROM ARRAY($asVolumes;$vlVolume)
      End if
      $vlVolume:=Find in array($asVolumes;"B:¥¥")
      If($vlVolume>0)
         DELETE FROM ARRAY($asVolumes;$vlVolume)
      End if
   End if
   $vlNbVolumes:=Size of array($asVolumes)
  ` For each volume
   For($vlVolume;1;$vlNbVolumes)
  ` Get the size, used space and free space
      gError:=0
      VOLUME ATTRIBUTES($asVolumes{$vlVolume};$vlSize;$vlUsed;$vlFree)
      If(gError=0)
  ` Is the free space large enough (plus an extra 32K) ?
         If($vlFree>=($1+32768))
  ` If so, check if the volume is unlocked...
            $vsDocName:=$asVolumes{$vlVolume}+Char(Directory symbol)+"XYZ"+String(Random)+".TXT"
```

```
            $vhDocRef:=Create document($vsDocName)
            If(OK=1)
               CLOSE DOCUMENT($vhDocRef)
               DELETE DOCUMENT($vsDocName)
 ` If everything's fine, return the name of the volume
               $0:=$asVolumes{$vlVolume}
               $vlVolume:=$vlNbVolumes+1
            End if
         End if
      End if
   End for
End if
ON ERR CALL("")
```

Once this project method is added to your application, you can for instance write:

```
$vsVolume:=Find volume for space(100*1024*1024)
If($vsVolume#"")
 ` Continue
Else
   ALERT("A volume with at least 100 MB of free space is required!")
End if
```

# VOLUME LIST

VOLUME LIST ( volumes )

| Parameter | Type | | Description |
|---|---|---|---|
| volumes | String array | ⇦ | Names of the volumes currently mounted |

## Description

The **VOLUME LIST** command populates the text array *volumes* with the names of the volumes currently defined (Windows) or mounted (Macintosh) on your machine.

- On Macintosh, it returns the list of the volumes visible at the Finder level. Only the names of the volumes are returned (for example "MacHD", "BootCamp", etc.).
- On Windows, it returns the list of the volumes currently defined whether or not each volume is physically present (i.e. the volume **E:¥** will be returned whether or not a CD or DVD is actually present in the drive). The names of the volumes are followed by the folder separator character ("C:¥").

## Example

Using a scrollable area named *atVolumes* you want to display the list of the volumes defined or mounted on your machine, you write:

```
Case of
   :(Form event=On Load)
      ARRAY TEXT(atVolumes;0)
      VOLUME LIST(atVolumes)

   //...
End case
```

# System Environment

- Count screens
- Current client authentication
- Current machine
- Current machine owner
- FONT LIST
- FONT STYLE LIST
- Gestalt
- GET SYSTEM FORMAT
- LOG EVENT
- Menu bar height
- Menu bar screen
- OPEN COLOR PICKER
- OPEN FONT PICKER
- PLATFORM PROPERTIES
- SCREEN COORDINATES
- SCREEN DEPTH
- Screen height
- Screen width
- Select RGB Color
- SET RECENT FONTS
- SET SCREEN DEPTH
- System folder
- Temporary folder
- *_o_Font name*
- *_o_Font number*

# Count screens

Count screens -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Number of monitors |

## Description

The **Count screens** command returns the number of screen monitors connected to your machine.

# Current client authentication

Current client authentication {( domain ; protocol )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| domain | Text | ⇐ | Domain name |
| protocol | Text | ⇐ | "Kerberos", "NTLM", or empty string |
| Function result | Text | ⇒ | Session user login returned by Windows |

## Description

The **Current client authentication** command asks the Windows Active Directory server to authenticate the current client and, if successful, returns the Windows login name for this client (session identifier). If the authentication failed, an empty string is returned.

This command can only be used in the context of an SSO implementation on Windows with 4D Server. For more information, please refer to the **Single Sign On (SSO) on Windows** section.

Usually, both the client and the server must be managed by the same Active Directory. However, different configurations can be supported, as described in the **Requirements for SSO** section.

The returned login string must be passed to your 4D identification module to grant access rights to the client based upon the Windows session login; if you managed to remove the 4D Server login dialog by setting a "Default user", you can implement an interface where the user does not need to reenter any IDs (see example).

Optionally, the command can return two text parameters:

- *domain*: name of domain to which the client belongs.
- *protocol*: name of protocol used by Windows to authenticate the user. It can be "Kerberos" or "NTLM", depending on available resources. If the authentication failed, an empty string ("") is returned.

These parameters can be used to accept or reject connections if you want to filter access with regard to the domain or protocol.

### Authentication security level

The security level of the authentication (i.e., how much you can trust the user login) depends on how the user has actually been identified. The value(s) returned in the **Current client authentication** command parameters will allow you to find out what the login (if any) is based on, and thus the security level:

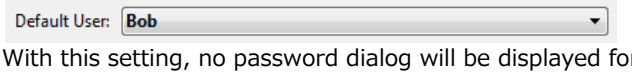
| Login | Domain | Protocol | Comments |
|---|---|---|---|
| Empty | Empty | Empty | Command was unable to get authentication information about the logged user |
| Filled | Empty | NTLM | ID returned is the local one, which has been defined on the local computer |
| Filled | Filled | NTLM | ID returned has been authenticated using the NTLM protocol in the Domain returned in the *domain* parameter. In this case, you must check the Domain to increase the security level. Since some architectures have a Domain forest, you need to make sure that the Domain where the user was authenticated was the expected one. |
| Filled | Filled | Kerberos | ID returned has been authenticated with the Kerberos protocol in the expected Domain. This configuration provides the highest level of security. |

For more information on these requirements, please refer to the paragraph.

## Example

In your 4D Server database, you have designed an access control system based on 4D's users and groups feature. You want to configure your application so that 4D remote users on Windows connect directly to 4D Server (no password dialog box is displayed), but while being logged with their actual rights:

1. In the "Security" page of the Database Settings dialog box, designate a user as the "default user":




With this setting, no password dialog will be displayed for a remote 4D that connects to the server; all clients being logged as "Bob".

2. In the On Server Open Connection database method, add the following code to check user authentication from the Active Directory:

```
 //On Server Open Connection database method
C_LONGINT($0;$1;$2;$3)
$login:=Current client authentication($domain;$protocol)
If($login #"") //a login was returned
 //call your custom authentication method
   $0:=CheckCredentials($login)
 //should return 0 in case of success or -1 for error
Else
   $0:=-1 //reject the connection
End if
```

**Note:** This example shows a basic scenario that must be adapted to your solutions. The 4D user custom authentication method (***CheckCredentials*** in the above example) could be based on one of the following implementations:

- replicate the Active directory names in the 4D user and group names, for an automatic mapping,
- map returned information to a custom [users] table,
- use LDAP features to get user credentials.

# Current machine

Current machine -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String | ➲ | Network name of the machine |

## Description

The **Current machine** command returns the name of the machine as set in the network parameters of the operating system.

## Example

Even if you are not running with the Client/Server version of the 4D environment, your application can include some network services that require your machine to be correctly configured. In the **On Startup database method** of your application, you write:

```
If((Current machine="")|(Current machine owner=""))
   ` Display a dialog box asking the user to setup the Network identity of his or her machine
End if
```

# Current machine owner

Current machine owner -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | String | Network name of machine owner |

## Description

The **Current machine owner** command returns the owner name of your machine, as set in the current user account on the machine.

## Example

See example for the **Current machine** command.

# FONT LIST

FONT LIST ( fonts {; listType | *} )

| Parameter | Type | | Description |
|---|---|---|---|
| fonts | Text array | ⇐ | Array of font names |
| listType \| * | Longint, Operator | ⇒ | Font type list to return or * to return font names under OS X |

## Description

The **FONT LIST** command populates the *fonts* text array with the names of scalable fonts available on your system.

The *listType* parameter lets you designate the type of font list you want to get. To do so, you can pass one of the following constants in the *listType* parameter, available in the "**Font Type List**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Favorite fonts | Longint | 1 | *fonts* contains the list of favorite fonts.<br>- Under Windows: list of active font family names.<br>- Under OS X: list of font family names found in the control panel, entitled "Favorites" in English, "Favoris" in French, "Favoriten" in German, and so on . This collection may be blank if the user has not added any favorite fonts. |
| Recent fonts | Longint | 2 | *fonts* contains the list of recent fonts (the ones used during the 4D session). This list is used in particular by multi-style text areas. |
| System fonts | Longint | 0 | *fonts* contains the list of all the system fonts. Default option when *listType* is omitted. |

Under OS X, when you pass the optional * parameter, the command populates the *fonts* array with the names of the fonts themselves, and not with the names of the font families. The default operation simplifies programmed management of rich text areas, which use font families. If you pass the * parameter, font names, for example, "Arial bold", "Arial italic", "Arial narrow italic," are returned instead of families, such as "Arial", "Arial black" or "Arial narrow".
Under Windows, the * parameter has no effect. The command still returns the font families.

**Note:** Under OS X, if you use the result of this command with the **ST SET ATTRIBUTES**, you must not pass the * parameter.

### About scalable fonts

This command returns only scalable fonts. Using non-scalable fonts (i.e. bitmap fonts) to design interfaces is not recommended since they are based on an outdated technology and suffer from limitations regarding size variations. They are not supported in cutting-edge features of 4D such as 4D Write Pro areas .

Under OS X, this principle has been in effect since OS X 10.4 (*QuickDraw* bitmap fonts are obsolete beginning with this version).

Under Windows, this principle is applied beginning with 4D v15 R4. In order to help developers select only modern fonts for their interfaces, only "trueType" or "openType" scalable fonts are listed. For example, "ASI_Mono", "MS Sans Serif" and "System" fonts are no longer available. In addition, GDI names are also ignored; only DirectWrite font family names are supported. For example, "Arial Black" or "Segoe UI Black" font families are not in the list; only "Arial" and "Segoe" are returned.

**Compatibility notes for Windows:**

- Bitmap fonts can still be used in your 4D forms (except in 4D Write Pro areas). They are just removed from the list returned by this command. However, to ensure compatibility with future versions of 4D and Windows, we recommend using only DirectWrite font families.
- Since bitmap fonts are filtered from the *fonts* parameter on Windows, the resulting list is different in 4D v15 R4 applications and higher, compared to previous releases. Please make sure to adapt your code if you were using this command to select a non-scalable font.

## Example 1

In a form, you want a drop-down list that displays a list of the fonts available on your system. The method of the drop-down list is as follows:

```
Case of
   :(Form event=On Load)
      ARRAY TEXT(asFont;0)
      FONT LIST(asFont)
   ...

End case
```

## Example 2

You want to get a list of recent fonts:

```
FONT LIST($arrFonts;Recent fonts)
```

# FONT STYLE LIST

FONT STYLE LIST ( fontFamily ; fontStyleList ; fontNameList )

| Parameter | Type | | Description |
|---|---|---|---|
| fontFamily | Text | ⇒ | Name of font family |
| fontStyleList | Text array | ⇐ | List of font styles supported by the font family |
| fontNameList | Text array | ⇐ | List of complete font names supported by the font family |

## Description

The **FONT STYLE LIST** command returns the list of font styles and the list of complete font names supported by the font family defined in the *fontFamily* parameter. This command allows you to design interfaces handling fonts and font styles, particularly in the context of 4D Write Pro areas (see **4D Write Pro Reference**).

In *fontFamily*, pass the name of the font family for which you want to know the supported font styles and names.

In *fontStyleList*, you pass a text array to be filled with the list of font styles supported by the *fontFamily*. Styles are returned using their localized names (i.e. an "Italic" element will be returned as "Itálico" on a Spanish system), so that you can build a localized "Style" pop-up menu, for example.

In *fontNameList*, you pass a text array to be filled with the complete list of font names supported by the *fontFamily*. Unlike the *fontStyleList* array, the *fontNameList* array returns non-localized values, i.e. font names based upon the system identification. Thus, font names will be independent from the system language. Elements of this array are strings intended to be used with the wk font 4D Write Pro attribute of the **WP SET ATTRIBUTES** command. Using this feature, 4D Write Pro documents can store font names and then be opened on machines using any system language without font issues.

If the *fontFamily* is not found on the machine, arrays are returned empty. To get the list of font families available on the machine, use the **FONT LIST** command.

## Example

You want to select styles of the "Verdana" font family (if available):

```
ARRAY TEXT($aTfonts;0)
ARRAY TEXT($aTStyles;0)
ARRAY TEXT($aTnames;0)
C_LONGINT($numStyle)

FONT LIST($aTfonts)
$numStyle:=Find in array($aTfonts;"Verdana")
If($numStyle#0)
   FONT STYLE LIST($aTfont{$numStyle};$aTStyles;$aTnames)
End if

 //For example, resulting arrays are:
 //$aTStyles{1}="Normal"
 //$aTStyles{1}="Italic"
 //$aTStyles{1}="Bold"
 //$aTStyles{1}="Bold Italic"

 // $aTnames{1}="Verdana"
 // $aTnames{1}="Verdana Italic"
 // $aTnames{1}="Verdana Bold"
 // $aTnames{1}="Verdana Bold Italic"
```

# Gestalt

Gestalt ( selector ; value ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| selector | String | ⇨ | 4-character gestalt selector |
| value | Longint | ⇦ | Gestalt result |
| Function result | Longint | ↩ | Error code result |

## Description

The **Gestalt** command returns in *value* a numeric value that denotes the characteristics of your system hardware and software, depending on the selector you pass in *selector*.

If the requested information is obtained, **Gestalt** returns 0 in *function result*; otherwise, it returns the error -5550. If the selector is unknown, **Gestalt** returns the error -5551.

**Important:** The Gestalt Manager is part of Mac OS. On Windows, some of the selectors are also implemented, but the usefulness of this command is limited.

For more information about the selectors that you can pass to Gestalt, refer to the Apple Developer documentation related to the Gestalt Manager, available on-line at the following address:
http://developer.apple.com/documentation/Carbon/Reference/Gestalt_Manager/index.html

## Example

Using version 10.4.11 of Mac OS, the following code displays the alert “You're running system version 0x1049”:

```
$vlErrCode:=Gestalt("sysv";$vlInfo)
If($vlErrCode=0)
   ALERT("You're running system version "+String($vlInfo;"&x"))
End if
```

# GET SYSTEM FORMAT

GET SYSTEM FORMAT ( format ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| format | Longint | ⇨ | System format to be retrieved |
| value | String | ⇦ | Value of format defined in the system |

## Description

The **GET SYSTEM FORMAT** command returns the current value of several regional parameters defined in the operating system. This command can be used to build “automatic” custom formats based on the system preferences.

In the *format* parameter, pass the type of parameter whose value you want to know. The result is returned directly by the system in the *value* parameter as a character string. In *format*, you must pass one of the following constants of the **System Format** theme. Below is a description of these constants:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Currency symbol | Longint | 2 | Currency symbol (e.g.: “$”) |
| Date separator | Longint | 13 | Separator used in date formats (e.g.: “/”) |
| Decimal separator | Longint | 0 | Decimal separator (e.g.: “.”) |
| Short date day position | Longint | 15 | Position of the day in the short date format: “1” = left, “2” = middle, “3” = right |
| Short date month position | Longint | 16 | Position of the month in the short date format: “1” = left, “2” = middle, “3” = right |
| Short date year position | Longint | 17 | Position of the year in the short date format: “1” = left, “2” = middle, “3” = right |
| System date long pattern | Longint | 8 | Long date display format in the form “dddd MMMM yyyy” |
| System date medium pattern | Longint | 7 | Medium date display format in the form “dddd MMMM yyyy” |
| System date short pattern | Longint | 6 | Short date display format in the form “dddd MMMM yyyy” |
| System time AM label | Longint | 18 | Additional label for a time before noon in 12-hour formats (e.g.: “Morning”) |
| System time long pattern | Longint | 5 | Long time display format in the form “HH:MM:SS” |
| System time medium pattern | Longint | 4 | Medium time display format in the form “HH:MM:SS” |
| System time PM label | Longint | 19 | Additional label for a time after noon in 12-hour formats (e.g.: “Afternoon”) |
| System time short pattern | Longint | 3 | Short time display format in the form “HH:MM:SS” |
| Thousand separator | Longint | 1 | Thousand separator (e.g.: “,”) |
| Time separator | Longint | 14 | Separator used in time formats (e.g.: “:”) |

## Example

On a check that is filled in mechanically, the amounts written are generally prefixed by “*” characters in order to prevent fraud. If the standard system display format for currency is “$ 5,422.33”, the format for checks should be of the type “$***5432.33”: no comma after the thousand digit and no space between the $ symbol and the first number. The format to be used with the **String** function must be “$*******.**”. To build it via programming, it is necessary to know the currency symbol and the decimal separator:

```
GET SYSTEM FORMAT(Currency symbol;$vCurrSymb)
GET SYSTEM FORMAT(Decimal separator;$vDecSep)
$MyFormat:="###"+$vCurrSymb+"*******"+$vDecSep+"**"
$Result:=String(amount;$MyFormat)
```

# LOG EVENT

LOG EVENT ( {outputType ;} message {; importance} )

| Parameter | Type | | Description |
|---|---|---|---|
| outputType | Longint | ➡ | Message output type |
| message | String | ➡ | Contents of the message |
| importance | Longint | ➡ | Message's importance level |

## Description

The **LOG EVENT** command sets up a customized system for recording internal events that occur during the use of your application.

Pass the custom information to be noted according to the event in *message*.

The optional *outputType* parameter specifies the output channel taken by the *message*. You can pass one of the following constants, located in the "**Log Events**" theme, in this parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Into 4D commands log | Longint | 3 | Indicates to 4D to record the *message* in the 4D commands log file, if this file has been activated. The 4D commands log file can be enabled using the **SET DATABASE PARAMETER** command (selector 34).<br>**Note:** 4D log files are grouped together in the **Logs** folder, which is created next to the database structure file (see the **Get 4D folder** command). |
| Into 4D debug message | Longint | 1 | Indicates to 4D to send the *message* to the system debugging environment. The result depends on the platform:<br>• Under Mac OS: the command sends the message to the Console<br>• Under Windows: the command sends the message as a debug message. To be able to read this message, you must have Microsoft Visual Studio or the DebugView utility for Windows (http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx) |
| Into 4D diagnostic log | Longint | 5 | Indicates to 4D to record the *message* in the 4D diagnostic file, if this file has been enabled.<br>The diagnostic log file can be enabled using the **SET DATABASE PARAMETER** command (selector 79). |
| Into 4D request log | Longint | 2 | Indicates to 4D to record the *message* in the 4D requests log, if this file has been activated |
| Into Windows log events | Longint | 0 | Indicates to 4D to send the *message* to the "Log events" of Windows. This log receives and stores messages coming from running applications. In this case, you can attribute a level of importance to *message* via the optional *importance* parameter (see below).<br>**Notes:**<br>• For this feature to be available, the Windows Log Events service must be running.<br>• Under Mac OS, the command does nothing with this output type |

If you do not pass the *outputType* parameter, the value 0 (Into Windows log events) is used by default.

If you have defined the *outputType* parameter as Into Windows log events, you can attribute a level of importance to *message*, via the optional *importance* parameter which helps you to read and understand the log events. There are three levels of importance: Information, Warning, and Error.
4D provides you with the following predefined constants, placed in the "**Log Events**" category:

| Constant | Type | Value |
|---|---|---|
| Error message | Longint | 2 |
| Information message | Longint | 0 |
| Warning message | Longint | 1 |

If you don’t pass anything in *importance* or pass an incorrect value, the default value (0) is used.

## Example

If you want to have keep track of when your database is opened under Windows, you could write the following line of code in the **On Startup database method**:

```
LOG EVENT(Into Windows log events;"The Invoice database was opened.")
```

Each time the database is opened, this information will be written in Windows’ log events and its level of importance will be 0.

# Menu bar height

Menu bar height -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ⮌ | Height (expressed in pixels) of menu bar (returns zero if menu bar is hidden) |

## Description

**Menu bar height** returns the height of the menu bar, expressed in pixels.

If the menu bar is hidden, the command returns 0.

# Menu bar screen

Menu bar screen -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Number of screen where menu bar is located |

## Description

**Menu bar screen** returns the number of the screen where the menu bar is located.

**Windows note:** On Windows, **Menu bar screen** usually returns 1.

# OPEN COLOR PICKER

OPEN COLOR PICKER {( textOrBackground )}

| Parameter | Type | | Description |
|---|---|---|---|
| textOrBackground | Longint | ⇒ | 0 or omitted = text color, 1 = text background color |

## Description

The **OPEN COLOR PICKER** command displays the system color picker dialog box.

**Note:** This is a modal dialog box under Windows but not under OS X.

When the user selects a color and validates the dialog box, this color is applied to the current text selection in the object with the focus, if the "Allow Font/Color Picker" property is checked for this object (see the *Design Reference* manual).

If you pass 0 in the *textOrBackground* parameter or omit this parameter, the selected color is applied to the text. If you pass 1 in *textOrBackground*, this color is applied to the text background.

If the color was changed, the On After Edit form event is generated for the object.

# OPEN FONT PICKER

OPEN FONT PICKER
Does not require any parameters

## Description

The **OPEN FONT PICKER** command displays the system font picker dialog box.

**Note:** This is a modal dialog box under Windows but not under OS X.

When the user selects a font and/or a style and validates the dialog box, the changes are applied to the current text selection in the object with the focus, if the "Allow Font/Color Picker" property is checked for this object (see the *Design Reference* manual). Otherwise, the command does nothing.

If the font was changed, the On After Edit form event is generated for the object .

## Example

In a form, you want to add a button to display the font picker in order to allow users to modify the font or style of a Text variable area. Make sure that:

- the Text variable has the "Allow font/color picker" property checked .
- the "Focusable" property for the button has been unchecked.

Here is the button code:

```
Case of
   :(Form event=On Clicked)
      GOTO OBJECT(textVar) //gives the variable the focus
      OPEN FONT PICKER
End case
```

# PLATFORM PROPERTIES

PLATFORM PROPERTIES ( platform {; system {; processor {; language}}} )

| Parameter | Type | | Description |
|---|---|---|---|
| platform | Longint | ⇐ | 2 = Mac OS, 3 = Windows |
| system | Longint | ⇐ | Depends on the version you are running |
| processor | Longint | ⇐ | Processor family |
| language | Longint | ⇐ | Depends on the system you are using |

## Description

The **PLATFORM PROPERTIES** command returns information about the type of operating system you are running, the version and the language of the operating system, and the processor installed on your machine.

**PLATFORM PROPERTIES** returns environment information in the *platform*, *system*, *processor* and *language* parameters.

*platform* indicates the operating system used. This parameter returns one the following predefined constants:

| Constant | Type | Value |
|---|---|---|
| Mac OS | Longint | 2 |
| Windows | Longint | 3 |

The information returned in *system* depends on the version of 4D you are running.

### Macintosh version

If you are running a Mac OS version of 4D, the *system* parameter returns a 32-bit (Long Integer) value, for which the high level word is unused and the low level word is structured like this:

- The high byte contains the major version number,
- The low byte is composed of two nibbles (4 bits each). The high nibble is the major update version number and the low nibble is the minor update version. Example: System 9.0.4 is coded as *$0904*, so you receive the decimal value *2308*.

**Note:** In 4D, you can extract these values using the % (modulo) and ¥ (integer division) **Numeric Operators** or the **Bitwise Operators**.

Use the following formula to find out the Mac OS main version number:

```
PLATFORM PROPERTIES($vlPlatform;$vlSystem)
$vlResult:=$vlSystem¥256
 //If $vlResult = 16 --> you are under Mac OS 10.x
 //If $vlResult # 16 --> you are under another Mac OS version
```

### Windows version

If you are running the Windows version of 4D, the *system* parameter returns a 32-bit (Long Integer) value, the bits and bytes of which are structured as follows:

If the high level bit is set to 0, it means you are running Windows NT, Windows 2000, Windows XP or Windows Vista. If the bit is set to 1, it means you are running a version of Windows that is too old.

**Note:** The high level bit fixes the sign of the long integer value. Therefore, in 4D, you just need to test the value returned by *system*; if it is negative, you are using an obsolete version of Windows. You can also use the **Bitwise Operators**.

The low byte gives the major Windows version number:

- If it returns 4, you are running Windows NT 4. If it returns 5, you are running Windows 2000, Windows Server 2003 or Windows XP (the sign of the value tells whether or not you are running NT/2000).
- If it returns 6, you are running Windows Vista, Windows Seven, or Windows 8.1.
- If it returns 10, you are running Windows 10. Note that in this case, the *system* parameter is also 10.

The next low byte gives the minor Windows version number.

**Note:** In 4D, you can extract these values using the % (modulo) and ¥ (integer division) **Numeric Operators** or the **Bitwise Operators**.

The *processor* parameter indicates the microprocessor "family" of the machine. Two values can be returned, available in the form of constants:

| Constant | Type | Value |
|---|---|---|
| Intel compatible | Longint | 586 |
| Power PC | Longint | 406 |

The combination of the *platform* and *processor* parameters can be used for example to know without ambiguity whether the machine used is of the “MacIntel” type (*platform=Mac OS* and *processor=Intel Compatible*).
The *language* parameter is used to find out the current language of the system on which the database is running. Here is a list of the codes that can be returned in this parameter, as well as their meanings:

| Code | Language |
|---|---|
| 1 | Arabic |
| 2 | Bulgarian |
| 3 | Catalan |
| 4 | Chinese |
| 5 | Czech |
| 6 | Danish |
| 7 | German |
| 8 | Greek |
| 9 | English |
| 10 | Spanish |
| 11 | Finnish |
| 12 | French |
| 13 | Hebrew |
| 14 | Hungarian |
| 15 | Icelandic |
| 16 | Italian |
| 17 | Japanese |
| 18 | Korean |
| 19 | Dutch |
| 20 | Norwegian |
| 21 | Polish |
| 22 | Portuguese |
| 24 | Romanian |
| 25 | Russian |
| 26 | Croatian |
| 26 | Serbian |
| 27 | Slovak |
| 28 | Albanian |
| 29 | Swedish |
| 30 | Thai |
| 31 | Turkish |
| 33 | Indonesian |
| 34 | Ukrainian |
| 35 | Belarusian |
| 36 | Slovenian |
| 37 | Estonian |
| 38 | Latvian |
| 39 | Lithuanian |
| 41 | Farsi |
| 42 | Vietnamese |
| 45 | Basque |
| 54 | Afrikaans |
| 56 | Faeroese |

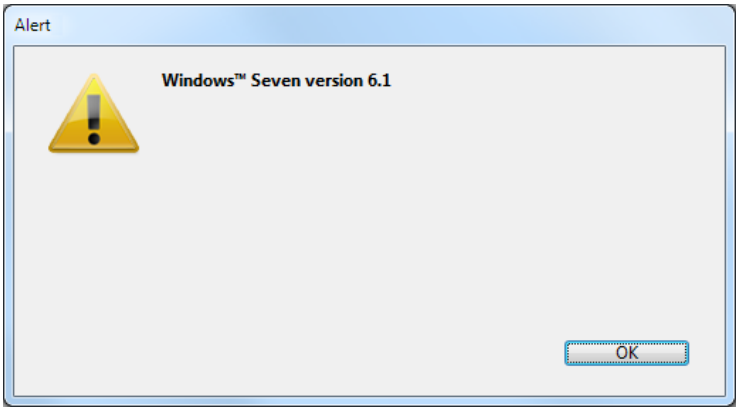
**Note:** If the command is not able to identify the system language, the value 9 (English) is returned.

## Example

The following project method displays an alert box showing the OS software you are using:

```
 //SHOW OS VERSION project method
PLATFORM PROPERTIES($vlPlatform;$vlSystem;$vlMachine)
If(($vlPlatform<2)|($vlPlatform>3))
   $vsPlatformOS:=""
Else
   If($vlPlatform=Windows)
      $vsPlatformOS:=""
      If($vlSystem<0)
         $vsPlatformOS:="Windows version too old"
      Else
         $winMajVers:=$vlSystem%256
         $winMinVers:=($vlSystem¥256)%256
         Case of
            :($winMajVers=4)
               $vsPlatformOS:="Windows™ NT"
            :($winMajVers=5)
               Case of
                  :($winMinVers=0)
                     $vsPlatformOS:="Windows™ 2000"
                  :($winMinVers=1)
                     $vsPlatformOS:="Windows™ XP"
                  :($winMinVers=2)
                     $vsPlatformOS:="Windows™ 2003"
                  Else
                     $vsPlatformOS:="Windows (undetermined version)"
               End case
            :($winMajVers=6)
               Case of
                  :($winMinVers=0)
                     $vsPlatformOS:="Windows™ Vista"
                  :($winMinVers=1)
                     $vsPlatformOS:="Windows™ Seven"
                  :($winMinVers=3)
                     $vsPlatformOS:="Windows™ 8.1"
                  Else
                     $vsPlatformOS:="Windows (undetermined version)"
               End case
            :($winMajVers=10) //$vlSystem=10 also
               $vsPlatformOS:="Windows™ 10"
         End case
      End if
      $vsPlatformOS:=$vsPlatformOS+" version "+String($winMajVers)+"."+String($winMinVers)
   Else
      $vsPlatformOS:="OS X version "
      If(($vlSystem¥256)=16)
         $vsPlatformOS:=$vsPlatformOS+"10"
      Else
         $vsPlatformOS:=$vsPlatformOS+String($vlSystem¥256)
      End if
      $vsPlatformOS:=$vsPlatformOS+"."+String(($vlSystem¥16)%16)+
        (("."+String($vlSystem%16))*Num(($vlSystem%16)#0))
   End if
End if
ALERT($vsPlatformOS)
```

On Windows, you get an alert box similar to this:




On Mac OS, you get an alert box similar to this:

# SCREEN COORDINATES

SCREEN COORDINATES ( left ; top ; right ; bottom {; screen} )

| Parameter | Type | | Description |
|---|---|---|---|
| left | Longint | ⇦ | Global left coordinate of screen area |
| top | Longint | ⇦ | Global top coordinate of screen area |
| right | Longint | ⇦ | Global right coordinate of screen area |
| bottom | Longint | ⇦ | Global bottom coordinate of screen area |
| screen | Longint | ⇨ | Screen number, or main screen if omitted |

## Description

The **SCREEN COORDINATES** command returns in *left*, *top*, *right,* and *bottom* the global coordinates of the screen specified by *screen*.

If you omit the *screen* parameter, the command returns the coordinates of the main screen.

# SCREEN DEPTH

SCREEN DEPTH ( depth ; color {; screen} )

| Parameter | Type | | Description |
|---|---|---|---|
| depth | Longint | ⇦ | Depth of the screen (number of colors = 2 ^ depth) |
| color | Longint | ⇦ | 1 = Color screen, 0 = Black and white or Gray scale |
| screen | Longint | ⇨ | Screen number, or main screen if omitted |

## Description

The **SCREEN DEPTH** command returns in *depth* and *color* information about the monitor.

The depth of the screen is returned in *depth*. The depth of the screen is the exponent of the power of 2 expressing the number of colors displayed on your monitor. For example, if your monitor is set for 256 colors (2^8), the depth of your screen is 8.

The following predefined constants are provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Black and white | Longint | 0 |
| Four colors | Longint | 2 |
| Millions of colors 24 bit | Longint | 24 |
| Millions of colors 32 bit | Longint | 32 |
| Sixteen colors | Longint | 4 |
| Thousands of colors | Longint | 16 |
| Two fifty six colors | Longint | 8 |

If the monitor is set to display in color, *1* is returned in *color*. If the monitor is set to display in gray scale, *0* is returned in *color*. Note that this value is significant on the Macintosh platform.

The following predefined constants are provided by 4D:

| Constant | Type | Value |
|---|---|---|
| Is color | Longint | 1 |
| Is gray scale | Longint | 0 |

The optional parameter *screen* specifies the monitor for which you want to get information. If you omit the *screen* parameter, the command returns the depth of the main screen.

## Example

Your application displays many color graphics. Somewhere in your database, you could write:

```
SCREEN DEPTH($vlDepth;$vlColor)
If($vlDepth<8)
   ALERT("The forms will look better if the monitor"+" was set to display 256 colors or more.")
End if
```

# Screen height

Screen height {( * )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | Windows: height of application window, or height of screen if * is specified Macintosh: height of main screen |
| Function result | Longint | ⇐ | Height expressed in pixels |

## Description

On Windows, **Screen height** returns the height of 4D application window (MDI window). If you specify the optional * parameter, the function returns the height of the screen.

On Macintosh, **Screen height** returns the height of the main screen, the screen where the menu bar is located.

# Screen width

Screen width {( * )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | Windows: width of application window, or width of screen if * is specified Macintosh: width of main screen |
| Function result | Longint | ⇐ | Width expressed in pixels |

## Description

On Windows, **Screen width** returns the width of 4D application window (MDI window). If you specify the optional * parameter, the function returns the width of the screen.

On Macintosh, **Screen width** returns the width of the main screen, the screen where the menu bar is located.

# Select RGB Color

Select RGB Color {( defaultColor {; message} )} -> Function result

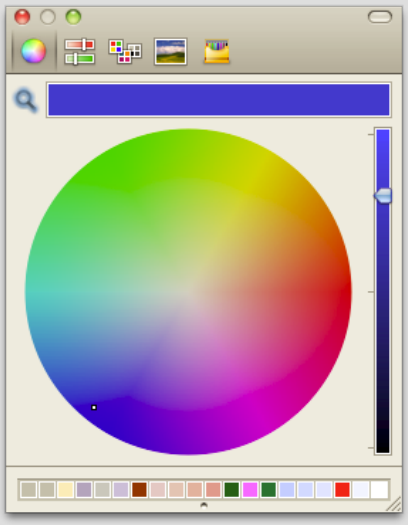
| Parameter | Type | | Description |
|---|---|---|---|
| defaultColor | Longint | ⇒ | Preselected RGB color |
| message | Alpha | ⇒ | Title of selection window |
| Function result | Longint | ⇐ | RGB color |

## Description

The **Select RGB Color** command displays the system color selection window and returns the RGB value of the color selected by the user.

The system color selection window appears as follows:

**Macintosh**



**Windows**


The optional *defaultColor* parameter preselects a color in the window. This parameter can be used, for example, to restore by default the last color set by the user. Pass an RGB-format color value in this parameter (for more information, refer to the description of the **OBJECT SET RGB COLORS** command). You can use one of the constants in the **SET RGB COLORS** theme. If the *defaultColor* parameter is omitted or if you pass 0, the color black is selected when the dialog box is opened.

The optional *message* parameter customizes the title of the system window. By default, if this parameter is omitted, the title “Colors” is displayed.

The effect of validating this dialog box differs depending on the platform:

- Under Windows, when the user clicks on **OK**, the command returns the value of the color selected in RGB format and the system variable *OK* is set to 1. If the user cancels the dialog box, the command returns -1 and the system variable *OK* is set to 0.
- Under Mac OS, you can only close this dialog box by clicking on the close box or by pressing on the **Esc** key. In both cases, the system variable *OK* is set to 1, regardless of the user actions in the window. The command returns the value of the color selected in RGB format. If the user did not select a color, the value returned is the one passed in *defaultColor* (if any) or 0 if *defaultColor* is not passed.

**Note:** This command must not be executed on the server machine nor within a Web process.

# SET RECENT FONTS

SET RECENT FONTS ( fontsArray )

| Parameter | Type | | Description |
|---|---|---|---|
| fontsArray | Text array | ➡ | Array of font names |

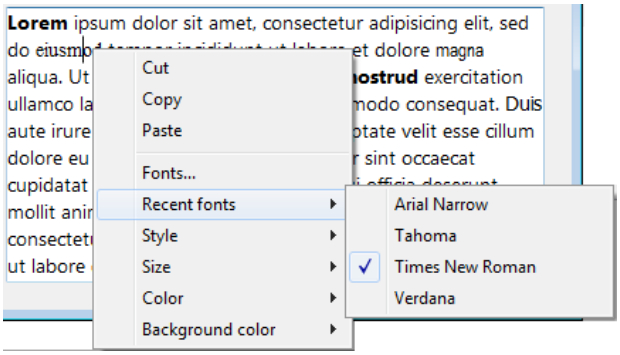
## Description

The **SET RECENT FONTS** command modifies the list of fonts displayed in the context menu of the "recent fonts".

This menu contains the names of the last fonts selected during the session. It is used in particular by **Programming Notes** areas.

## Example

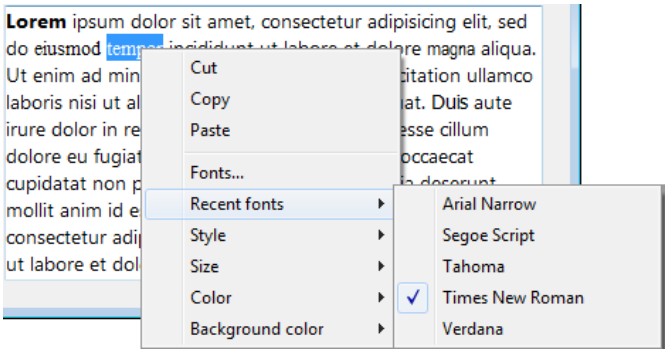
You want to add a font to the menu of recent fonts:




You execute the following code:

```
ARRAY TEXT($arrRecent;0)
FONT LIST($arrRecent;2)
APPEND TO ARRAY($arrRecent;"Segoe Script")
APPEND TO ARRAY($arrRecent)
```

Then the menu contains:



# SET SCREEN DEPTH

SET SCREEN DEPTH ( depth {; color {; screen}} )

| Parameter | Type | | Description |
|---|---|---|---|
| depth | Longint | ⇒ | Depth of the screen (number of colors = 2 ^ Screen depth) |
| color | Longint | ⇒ | 1 = Color, 0 = Gray Scale |
| screen | Longint | ⇒ | Screen number, or main screen if omitted |

## Description

**SET SCREEN DEPTH** changes the depth and color/gray scale settings of the screen whose number you pass in *screen*. If you omit this parameter, the command is applied to the main screen.

For details about the values you pass in *color* and *depth*, see the description of the command **SCREEN DEPTH**.

# System folder

System folder {( type )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| type | Longint | ⇒ | Type of system folder |
| Function result | String | ⇐ | Pathname to a system folder |

## Description

The **System folder** command returns the pathname to a particular folder of the operating system or to the active Windows or Mac OS System folder itself.

You pass in the optional *type* parameter a value indicating the type of system folder. 4D provides you with the following predefined constants, placed in the "**System Folder**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Applications or program files | Longint | 16 | |
| Desktop | Longint | 15 | |
| Documents folder | Longint | 17 | "Documents" folder of user |
| Favorites Win | Longint | 14 | |
| Fonts | Longint | 1 | |
| Start menu Win_all | Longint | 8 | |
| Start menu Win_user | Longint | 9 | |
| Startup Win_all | Longint | 4 | |
| Startup Win_user | Longint | 5 | |
| System | Longint | 0 | |
| System Win | Longint | 12 | |
| System32 Win | Longint | 13 | |
| User preferences_all | Longint | 2 | |
| User preferences_user | Longint | 3 | |

**Notes:**

- The constants suffixed **Win** can be used on Windows only. When they are used on Mac OS, **System folder** will return an empty string.
- The pathnames to some system folders can specific to the current user. Constants 2 to 9 allow you to choose whether you want to obtain the pathname to a folder which is shared by all users, or customized for the current user.

If you omit the *type* parameter, the function will return the pathname to active System folder (= constant System).

# Temporary folder

Temporary folder -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String | | Pathname to temporary folder |

## Description

The **Temporary folder** command returns the pathname to the current temporary folder set by your system.

## Example

See example for the **APPEND DATA TO PASTEBOARD** command.

# _o_Font name

_o_Font name ( fontNumber ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| fontNumber | Longint | ⇒ | Font number for which to return the font name |
| Function result | String | ⇐ | Font name |

## Description

*This command is obsolete and must no longer be used beginning with 4D v14. It is kept for compatibility reasons but it will not be supported in future versions of the program.*

# _o_Font number

_o_Font number ( fontName ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| fontName | String | ⇒ | Font name for which to return the font number |
| Function result | Longint | ⇒ | Font number |

## Description

*This command is obsolete and must no longer be used beginning with 4D v14. It is kept for compatibility reasons but it will not be supported in future versions of the program.*

# Table

- Current default table
- Current form table
- DEFAULT TABLE
- NO DEFAULT TABLE

# Current default table

Current default table -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Pointer | Pointer to the default table |

## Description

**Current default table** returns a pointer to the table that has been passed to the last call to **DEFAULT TABLE** for the current process.

## Example

Provided a default table has been set, the following line of code sets the window title to the name of the current default table:

```
SET WINDOW TITLE(Table name(Current default table))
```

# Current form table

Current form table -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Pointer | Pointer to the table of the currently displayed form |

## Description

The **Current form table** command returns the pointer to the table of the form being displayed or printed in the current process.

The function returns **Nil** in the following cases:

- There is no form being displayed or printed in the current process,
- The current form is a project form.

If there are several windows open for the current process (which means that the window opened last is the current active window), the command returns the pointer to the table of the form displayed in the active window.

If the currently displayed form is the Detail form for a subform area, you are in data entry and you double-clicked on a record or a subrecord of a double-clickable subform area. In this case, the command returns:

- The pointer to the table shown in the subform area, if the subform displays a table.
- A non-significant pointer, if the subform area displays a subtable.

## Example

Throughout your application, you use the following convention when displaying a record:
If the variable *vsCurrentRecord* is present in a form, it displays “New Record” if you are working with a new record. If you are working with the 56th record of a selection composed of 5200 records, it displays “56 of 5200”.

To do so, use the object method to create the variable *vsCurrentRecord*, then copy and paste it into all of your forms:

```
  ` vsCurrentRecord non-enterable variable object method
Case of
   :(Form event=On Load)
      C_STRING(31;vsCurrentRecord)
      C_POINTER($vpParentTable)
      C_LONGINT($vlRecordNum)
      $vpParentTable:=Current form table
      $vlRecordNum:=Record number($vpParentTable->)
      Case of
         :($vlRecordNum=-3)
            vsCurrentRecord:="New Record"
         :($vlRecordNum=-1)
            vsCurrentRecord:="No Record"
         :($vlRecordNum>=0)
            vsCurrentRecord:=String(Selected record number($vpParentTable->))+" of "+
            String(Records in selection($vpParentTable->))
      End case
End case
```

# DEFAULT TABLE

DEFAULT TABLE ( aTable )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table to set as the default |

## Description

**Tip:** Although using **DEFAULT TABLE** and omitting the table name may make the code easier to read, many programmers find that using this command actually causes more problems and confusion than it is worth. In particular, note that **DEFAULT TABLE** takes priority when you use, for example, the **DIALOG** command with a project form and there is a default table form with the same name.

**DEFAULT TABLE** sets *aTable* as the default table for the current process.

There is no default table for a process until the **DEFAULT TABLE** command is executed. After a default table has been set, any command that omits the *table* parameter will operate on the default table. For example, consider this command:

```
FORM SET INPUT([Table];"form")
```

If the default table is first set to [Table], the same command could be written this way:

```
FORM SET INPUT("form")
```

One reason for setting the default table is to create code that is not table specific. Doing this allows the same code to operate on different tables. You can also use pointers to tables to write code that is not table specific. For more information about this technique, see the description of the **Table name** command.

**DEFAULT TABLE** does not allow the omission of table names when referring to fields. For example:

```
[My Table]My Field:="A string" ` Good
```

could not be written as:

```
DEFAULT TABLE([My Table])
My Field:="A string" ` WRONG
```

because a default table had been set. However, you can omit the table name when referring to fields in the table method, form, and objects that belong to the table.

In 4D, all tables are "open" and ready for use. **DEFAULT TABLE** does not open a table, set a current table, or prepare the table for input or output. **DEFAULT TABLE** is simply a programming convenience to reduce the amount of typing and make the code easier to read.

## Example

The following example first shows code without the **DEFAULT TABLE** command. It then shows the same code, with **DEFAULT TABLE**. The code is a loop commonly used to add new records to a database. The **FORM SET INPUT** and **ADD RECORD** commands both require a table as the first parameter:

```
FORM SET INPUT([Customers];"Add Recs")
Repeat
   ADD RECORD([Customers])
Until(OK=0)
```

Specifying the default table results in this code:

```
DEFAULT TABLE([Customers])
FORM SET INPUT("Add Recs")
Repeat
   ADD RECORD
Until(OK=0)
```

# NO DEFAULT TABLE

NO DEFAULT TABLE
Does not require any parameters

## Description

The **NO DEFAULT TABLE** command is used to cancel the effect of the **DEFAULT TABLE** command. After this command is executed, there is no longer any default table defined for the process.
This command will have no effect if the **DEFAULT TABLE** command has not been called beforehand.

This command concerns the use of project forms (forms not linked with tables): most of the commands related to forms (apart from user forms) accept an optional *aTable* parameter as their first parameter. For example, this is the case with the **FORM GET PARAMETER**, **Open form window** or **DIALOG** commands. Since a project form and table form can have the same name, this parameter can be used to determine the form to be used: pass the *aTable* parameter when you want to target the table form and omit it in the case of a project form.

In a database containing a project form named “TheForm” and a table form with the same name for the [Table1] table:

```
DIALOG([Table1];"TheForm") `4D uses the table form
DIALOG("TheForm") `4D uses the project form
```

However, this principle is null and void if the **DEFAULT TABLE** command is executed when the database contains a project form and a table form with the same name. In fact, in this case 4D will use the table form by default, even if the *aTable* parameter is not passed. In order to guarantee the use of project forms, simply use the **NO DEFAULT TABLE** command.

## Example

In a database containing a project form named “TheForm” and a table form with the same name for the [Table1] table:

```
DEFAULT TABLE([Table1])
DIALOG("TheForm") `4D uses the table form
NO DEFAULT TABLE
DIALOG("TheForm") `4D uses the project form
```

# Tools

- BASE64 DECODE
- BASE64 ENCODE
- Choose
- Generate digest
- Generate UUID
- GET ACTIVITY SNAPSHOT
- GET MACRO PARAMETER
- LAUNCH EXTERNAL PROCESS
- OPEN URL
- PROCESS 4D TAGS
- SET ENVIRONMENT VARIABLE
- SET MACRO PARAMETER

# BASE64 DECODE

BASE64 DECODE ( {encodedText ;} blob )

| Parameter | Type | | Description |
|---|---|---|---|
| encodedText | Text | ⇒ | Text containing BLOB encoded in Base64 format |
| blob | BLOB | ⇒ | BLOB encoded in Base64 format |
| | | ⇐ | Decoded BLOB |

## Description

The **BASE64 DECODE** command allows you to decode the BLOB coded in Base64 format passed in the *encodedText* or *blob* parameter.

If you pass the *encodedText* parameter, the command decodes its contents and returns it in the *blob* parameter. It must contain a BLOB encoded in Base64 format. In this case, the initial contents of the *blob* parameter are ignored by the command.
If you omit the *encodedText* parameter, the command directly modifies the BLOB passed in the *blob* parameter.

The command does not verify the contents of the *encodedText* or *blob* parameter. You must verify that the data passed is actually coded in Base64 format, otherwise the result will be incorrect.

## Example

This example lets you transfer a picture via a BLOB:

```
C_BLOB($sourceBlob)
C_PICTURE($mypicture)
$mypicture:=[people]photo
PICTURE TO BLOB($mypicture;$sourceBlob;".JPG")
C_TEXT($base64Text)
BASE64 ENCODE($sourceBlob;$base64Text) //Encoding of text
 // the binary is now available as character strings in $base64Text

C_TEXT($base64Text)
C_BLOB($targetBlob)
BASE64 DECODE($base64Text;$targetBlob) //Decoding of text
 // the binary encoded in base 64 is now available as a BLOB in $blobTarget
```

# BASE64 ENCODE

BASE64 ENCODE ( blob {; encodedText} )

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ⇒ | BLOB to encode in Base64 format |
| | | ⇐ | BLOB encoded in Base64 format |
| encodedText | Text | ⇐ | Result of BLOB encoded in Base64 format |

## Description

The **BASE64 ENCODE** command encodes the BLOB passed in the *blob* parameter in Base64 format.

If you pass the *encodedText* parameter, it receives the encoded contents of the *blob* as text at the end of command execution. In this case, the *blob* parameter itself is not modified by the command.
If you omit the *encodedText* parameter, the command directly modifies the BLOB passed as a parameter.

Base64 encoding modifies 8-bit coded data so that they do not keep more than 7 useful bits. This encoding is required, for example, for handling BLOBs using XML.

# Choose

Choose ( criterion ; value {; value2 ; ... ; valueN} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| criterion | Boolean, Integer | ⇒ | Value to test |
| value | Expression | ⇒ | Possible values |
| Function result | Expression | ⇐ | Value of criterion |

## Description

The **Choose** command returns one of the values passed in the *value1*, *value2*, etc. parameters depending on the value of the *criterion* parameter.

You can pass either a Boolean or Number type in the *criterion* parameter:

- If *criterion* is a Boolean, **Choose** returns *value1* if the Boolean equals True and *value2* if the Boolean equals False. In this case, the command expects exactly three parameters: *criterion*, *value1* and *value2*.
- If *criterion* is an integer, **Choose** returns the value whose position corresponds to *criterion*. Be careful, numbering of the values begins with 0 (the position of *value1* is thus 0). In this case, the command expects at least two parameters: *criterion* and *value1*.

The command accepts all types of data for the *value* parameter(s), except for pictures, pointers, BLOBS and arrays. Nevertheless, you need to make sure that all the values passed are of the same type, 4D will not carry out any verification on this point.

If no *value* corresponds to *criterion*, **Choose** returns a "null" value with respect to the type of the *value* parameter (for example, 0 for a Number type, "" for a String type, and so on).

This command can be used to generate concise code that replaces tests of the "Case of" type that take up several lines (see example 2). It is also very useful in places where formulas can be executed: query editor, application of a formula, quick report editor, column calculated in a listbox, and so on.

## Example 1

Here is an example of the typical use of this command with a Boolean type criterion:

```
vTitle:=Choose([Person]Masculine;"Mr";"Ms")
```

This code is strictly equivalent to:

```
If([Person]Masculine)
   vTitle:="Mr"
Else
   vTitle:="Ms"
End if
```

## Example 2

Here is an example of the typical use of this command with a Number type criterion:

```
vStatus:=Choose([Person]Status;"Single";"Married";"Widowed";"Divorced")
```

This code is strictly equivalent to:

```
Case of
   :([Person]Status=0)
      vStatus:="Single"
   :([Person]Status=1)
      vStatus:="Married"
   :([Person]Status=2)
      vStatus:="Widowed"
```

```
    :([Person]Status=3)
       vStatus:="Divorced"
End case
```

# Generate digest

Generate digest ( param ; algorithm ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| param | BLOB, Text variable | ➡ | Blob or text for which to get digest key |
| algorithm | Longint | ➡ | Algorithm used to return key:<br>0 = MD5 Digest, 1 = SHA1 Digest, 2 = 4D digest |
| Function result | Text | ⬅ | Value of digest key |

## Description

The **Generate digest** command returns the digest key of a BLOB or text after application of an encryption algorithm.

In 4D, the following algorithms are available: **MD5** (*Message Digest 5*)**, SHA-1** (*Secure Hash 1*) and **4D** (internal algorithm). These algorithms are different hash functions:

- MD5 is a series of 16 bytes returned as a string of 32 hexadecimal characters.
- SHA-1 is a series of 20 bytes returned as a string of 40 hexadecimal characters.
- 4D designates the internal algorithm used by 4D to encrypt user passwords. This algorithm is particularly useful in the context of the **On 4D Mobile Authentication database method** when you want to use your own list of users.

The value returned for the same object is the same on all the platforms (Mac/Windows, 32 or 64 bits). The calculation is performed based on the representation in UTF-8 of the text passed in the parameter.

**Note:** If you use the command with an empty text/BLOB, it does not return void but returns the following value: "d41d8cd98f00b204e9800998ecf8427e" (MD5) or "da39a3ee5e6b4b0d3255bfef95601890afd80709" (SHA-1).

Pass a Text or BLOB field or variable in the *param* parameter. The **Generate digest** function returns the digest key as a string.

In the *algorithm* parameter, pass a value designating which hash function to use. Use one of the following constants, found in the **Digest Type** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| 4D digest | Longint | 2 | Use internal algorithm of 4D |
| MD5 digest | Longint | 0 | Use the MD5 algorithm |
| SHA1 digest | Longint | 1 | Use the SHA-1algorithm |

If the calculation of the digest key is not performed correctly, the function generates an error that you can intercept using the **ON ERR CALL** command and the function returns an empty string.

## Example 1

This example compares two documents using the MD5 algorithm:

```
PLATFORM PROPERTIES($Platf;$Syst;$vlMachine)
 // Open the first document as read-only
$Same:=True
$vhDocRef1:=Open document("";"*";Read Mode))
If(OK=1) // If a document is selected
   DOCUMENT TO BLOB(Document;$FirstBlob) // Load document
   If(OK=1)
      If($Platf=Mac OS)
         DOCUMENT TO BLOB(Document;$FirstBlobRF;*)
 // Under Mac OS, load resource fork
         $MD5_1RF:=Generate digest($FirstBlobRF;MD5 digest)
      End if

 // Open the second document as read-only
      $vhDocRef2:=Open document("";"*";Read Mode))
      If(OK=1)
         DOCUMENT TO BLOB(Document;$SecondBlob)
         If(OK=1)
```

```
            If($Platf=Mac OS)
               DOCUMENT TO BLOB(Document;$SecondBlobRF;*)
               $MD5_2RF:=Generate digest($SecondBlobRF;MD5 digest)
               If($MD5_1RF#$MD5_2RF) // Compare digests
                  $Same:=False
               End if
            End if
            $MD5_1:=Generate digest($FirstBlob;MD5 digest)
            $MD5_2:=Generate digest($SecondBlob;MD5 digest)
            If(($MD5_1#$MD5_2)|($Same=False))
               ALERT("These two documents are different.")
            End if
         End if
      End if
   End if
End if
```

## Example 2

These examples illustrate how to retrieve the digest key of a text:

```
$key1:=Generate digest("The quick brown fox jumps over the lazy dog.";MD5 digest)
 // $key1 is "e4d909c290d0fb1ca068ffaddf22cbd0"
$key2:=Generate digest("The quick brown fox jumps over the lazy dog.";SHA1 digest)
 // $key2 is "408d94384216f890ff7a0c3528e8bed1e0b01621"
```

## Example 3

This example only accepts the "admin" user with the password "123" that does not match a 4D user:

```
 //On REST Authentication database method
C_TEXT($1;$2)
C_BOOLEAN($0;$3)
 //$1: user
 //$2: password
 //$3: digest mode
If($1="admin")
   If($3)
      $0:=($2=Generate digest("123";4D digest))
   Else
      $0:=($2="123")
   End if
Else
   $0:=False
End if
```

# Generate UUID

Generate UUID -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | String | ➲ | New UUID as non-canonical text (32 characters) |

## Description

The **Generate UUID** returns a new 32-character UUID identifier in non-canonical form.

An UUID is a 16-byte number (128 bits). It contains 32 hexadecimal characters. It can be expressed either in non-canonical form (series of 32 letters [A-F, a-f] and/or numbers [0-9], for example 550e8400e29b41d4a716446655440000) or in canonical form (groups of 8,4,4,4,12, for example 550e8400-e29b-41d4-a716-446655440000).

In 4D, UUID numbers can be stored in fields. For more information, please refer to the section of the *Design Reference* manual.

## Example

Generation of a UUID in a variable:

```
C_TEXT(MyUUID)
MyUUID:=Generate UUID
```

# GET ACTIVITY SNAPSHOT

GET ACTIVITY SNAPSHOT ( arrActivities | arrUUID ; arrStart ; arrDuration ; arrInfo {; arrDetails}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| arrActivities \| arrUUID | Object array, Text array | ⇦ | Complete description of operations (object array) or Operation UUIDs (text array) |
| arrStart | Text array | ⇦ | Operation start times |
| arrDuration | Longint array | ⇦ | Operation durations in seconds |
| arrInfo | Text array | ⇦ | Description |
| arrDetails | Object array | ⇦ | Details of context and sub-operations (if any) |
| * | Operator | ⇨ | If passed = Get server activity |

## Description

The **GET ACTIVITY SNAPSHOT** command returns a single or several arrays describing operations in progress on the 4D data. These operations usually display a progress window.

This command is used to get a snapshot of the *x* operations that are most time-consuming and/or run most frequently, such as cache writing or the execution of formulas.

**Note:** The information returned by the **GET ACTIVITY SNAPSHOT** command is the same as that displayed on the "Real Time Monitor" (RTM) page of the 4D Server administration window (see *4D Server Reference Guide*).

By default, **GET ACTIVITY SNAPSHOT** processes operations performed locally (with 4D single-user, 4D Server or 4D in remote mode). However, with 4D in remote mode, you can also get a snapshot of operations performed on the server: you just need to pass the asterisk (*) as the last parameter. In this case, the server data is recovered locally.
The * parameter is ignored when the command is executed on 4D Server or 4D single-user.

The **GET ACTIVITY SNAPSHOT** command accepts two syntaxes:

- syntax using only an object array.
- syntax using several arrays.

### First syntax: GET ACTIVITY SNAPSHOT ( arrActivities {; *})

With this syntax, all the operations are returned in a structured form in the 4D object array (*arrActivities*). Each element of the array is an object built as follows:

```
[
    {
        "message":"xxx",
        "maxValue":12321,
        "currentValue":63212,
        "interruptible:0,
        "remote":0,
        "uuid":"deadbeef",
        "taskId":xxx,
        "startTime":"2014-03-20 13:37:00:123",
        "duration":92132,
        "dbContextInfo":{
            "task_id": xxx,
            "user_name": Jean,
            "host_name": HAL,
            "task_name": "CreateIndexLocal",
            "client_uid": "DE4DB33F33F"
            "user4d_id ": 1,
            "client_version ": 123456
        },
        "dbOperationDetails":{
            table: "myTable"
            field: "Field_1"
        },
        "subOperations":[
            {"message":"xxx",
            ...}
```

```
            ]
        },
        {...}
    ]
```

Here is a description of each property returned:

- *message* (text): label of operation
- *maxValue* (number): number of iterations set for the operation (-1 if non-iterative operation)
- *currentValue* (number): current iteration
- *interruptible* (number): operation can be interrupted by user (0=true, 1=false)
- *remote* (number): operation paired between client and server (0=true, 1=false)
- *uuid* (text): UUID identifier of operation
- *taskId* (number): Internal identifier of the process at the origin of the operation
- *startTime* (text): start time of operation in the "yyyy:mm:dd hh:mm:ss:mls" format
- *duration* (number): duration of operation in milliseconds
- *dbContextInfo* (objet): information concerning operations handled by the database engine. Contains the following properties:
  - *host_name* (string): name of host that launched the operation
  - *user_name* (string): name of 4D user whose session launched the operation
  - *task_name* (string): name of process that launched the operation
  - *task_id* (num): ID number of process that launched the operation
  - *client_uid* (string): optional, uuid of client that launched the operation
  - *is_remote_context* (boolean, 0 or 1): optional, indicates whether the database operation was launched by a client (value 1) or by the server through a stored procedure (value 0)
  - *user4d_id* (num): ID number of the current 4D user on the client side
  - *client_version* (string): four digits representing the version of the 4D engine of the application, as returned by the **Application version** command.

  **Note:** client_uid and is_remote_context are only available in client/server mode. client_uid is only returned when the database operation was started on a client machine.
- *dbOperationDetails* (object): property returned only when the operation calls the database engine (this is the case, for instance, for queries and sorts). This is an object containing specific information related to the operation itself. The properties available depend on the type of database operation performed. More specifically, these properties include:
  - table (string): name of table involved in the operation
  - field (string): name of field involved in the operation
  - queryPlan (string): query plan defined for the operation
  - ...
- *subOperations* (array): array of objects containing sub-operations of the current operation (if any). The structure of each sub-element is identical to the one in the main object. If the current operation does not have any sub-operations, then *subOperations* is empty.

### Second syntax: GET ACTIVITY SNAPSHOT ( arrUUID ; arrStart ; arrDuration ; arrInfo {;arrSubOp} {; *})

With this syntax, all the operations are returned in several synchronized arrays (each operation causes an element to be added to all of the arrays). The following arrays are returned:

- *arrUUID*: contains the UUIDs for each operation (corresponds to the *uuid* property of the *arrActivities* object in the previous syntax).
- *arrStart*: contains the start times for each operation (corresponds to the *startTime* property of the *arrActivities* object).
- *arrDuration*: contains the durations of each operation in milliseconds (corresponds to the *duration* property of the *arrActivities* object).
- *arrInfo* : contains the labels describing each operation (corresponds to the *message* property of the *arrActivities* object).
- *arrDetails* (optional): each element of this array is an object containing the following properties:
  - *"dbContextInfo"* (object): see above
  - *"dbOperationDetails"* (object): see above
  - "subOperations". The value of this property is an object array containing all the sub-operations for the current operation. If the current operation does not have any sub-operations, the value of the *subOperations* property is an empty array (corresponds to the *subOperations* property of the *arrActivities* object)

## Example

This method, executed in a separate process on 4D or 4D Server, provides a snapshot of the operations that are underway:

```
ARRAY TEXT(arrUUID;0)
ARRAY TEXT(arrStart;0)
ARRAY LONGINT(arrDuration;0)
ARRAY TEXT(arrInfo;0)

Repeat
   GET ACTIVITY SNAPSHOT(arrUUID;arrStart;arrDuration;arrInfo)
   If(Size of array(arrUUID)>0)
      TRACE // calling of debugger
   End if
Until(False) // Infinite loop
```

You get arrays such as:

| Expression | Value |
|---|---|
| arrUUID | 6 elements |
| arrUUID | 0 |
| arrUUID{0} | "" |
| arrUUID{1} | "1858EB64D281A7429E2012CEEE78499A" |
| arrUUID{2} | "07BBF193B1C0184EA26F1D4235A89CE6" |
| arrUUID{3} | "AE18AFA9426B424FBFB1575554751E05" |
| arrUUID{4} | "715C5B028868E44BAC4062AB0B507601" |
| arrUUID{5} | "BEF371C7CFF45946A6B5FE3488692BD4" |
| arrUUID{6} | "5C51ED352AF1344AA3895D8236D9EC25" |
| arrStart | 6 elements |
| arrStart | 0 |
| arrStart{0} | "" |
| arrStart{1} | "12/3/2013 - 11:32:34" |
| arrStart{2} | "12/3/2013 - 11:32:35" |
| arrStart{3} | "12/3/2013 - 11:32:34" |
| arrStart{4} | "12/3/2013 - 11:32:34" |
| arrStart{5} | "12/3/2013 - 11:32:34" |
| arrStart{6} | "12/3/2013 - 11:32:35" |
| arrDuration | 6 elements |
| arrDuration | 0 |
| arrDuration{0} | 0 |
| arrDuration{1} | 5747 |
| arrDuration{2} | 5388 |
| arrDuration{3} | 5752 |
| arrDuration{4} | 5653 |
| arrDuration{5} | 5959 |
| arrDuration{6} | 5466 |
| arrInfo | 6 elements |
| arrInfo | 0 |
| arrInfo{0} | "" |
| arrInfo{1} | "Array to selection: 9 of 100" |
| arrInfo{2} | "Loading data" |
| arrInfo{3} | "Array to selection: 7 of 100" |
| arrInfo{4} | "Sequential searching on Companies: 11 of 98167 records" |
| arrInfo{5} | "Deleting records: 6 of 10" |
| arrInfo{6} | "Sequential searching on Companies: 13 of 98167 records" |

# GET MACRO PARAMETER

GET MACRO PARAMETER ( selector ; textParam )

| Parameter | Type | | Description |
|---|---|---|---|
| selector | Longint | ⇨ | Selection to use |
| textParam | Text | ⇦ | Returned text |

## Description

The **GET MACRO PARAMETER** command returns, in the *paramText* parameter, all or part of the text of the method from which it was called.

The *selector* parameter can be used to set the type of information to be returned. You can pass one of the following constants, added to the "**4D Environment**" theme:

| Constant | Type | Value |
|---|---|---|
| Full method text | Longint | 1 |
| Highlighted method text | Longint | 2 |

If you pass Full method text in *selector*, all of the text of the method will be returned in *paramText*. If you pass Highlighted method text in *selector*, only the text selected in the method will be returned in *paramText*.

## Example

Refer to the example of the **SET MACRO PARAMETER** command.

# LAUNCH EXTERNAL PROCESS

LAUNCH EXTERNAL PROCESS ( fileName {; inputStream {; outputStream {; errorStream}}}{; pid} )

| Parameter | Type | | Description |
|---|---|---|---|
| fileName | String | ⇒ | File path and arguments of file to launch |
| inputStream | String, BLOB | ⇒ | Input stream (stdin) |
| outputStream | String, BLOB | ⇐ | Output stream (stdout) |
| errorStream | String, BLOB | ⇐ | Error stream (stderr) |
| pid | Longint | ⇐ | Unique identifier for external process |

## Description

The **LAUNCH EXTERNAL PROCESS** command launches an external process from 4D under Mac OS X and Windows.

Under Mac OS X, this command provides access to any executable application that can be launched from the Terminal.

Pass the fixed file path of the application to execute, as well as any required arguments (if necessary), in the *fileName* parameter.

Under Mac OS X, you can also pass the application name only; 4D will then use the **PATH** environment variable to locate the executable.

**Warning:** This command can only launch executable applications; it cannot execute instructions that are part of the shell (command interpreter). For example, under Mac OS it is not possible to use this command to execute the *echo* instruction or indirections.

The *inputStream* parameter (optional) contains the *stdin* of the external process. Once the command has been executed, the *outputStream* and *errorStream* parameters (if passed) return respectively the *stdout* and *stderr* of the external process. You can use BLOB parameters instead of strings if you are working with binary data (such as pictures).

**Note:** If you use the *_4D_OPTION_BLOCKING_EXTERNAL_PROCESS* environment variable via the **SET ENVIRONMENT VARIABLE** command (asynchronous execution), the *outputStream* and *errorStream* parameters are not returned.

When passed, the *pid* parameter (longint) returns the system level ID for the process created to launch the command, regardless of the *_4D_OPTION_BLOCKING_EXTERNAL_PROCESS* option status. With this information, it is easier to interact with a created external process thereafter, e.g. to stop it. If the process launch fails, the *pid* parameter is not returned.

## Examples under Mac OS X

The following examples use the Mac OS X Terminal available in the Application/Utilities folder.

1. To change permissions for a file (*chmod* is the Mac OS X command used to modify file access):

```
LAUNCH EXTERNAL PROCESS("chmod +x /folder/myfile.txt")
```

2. To edit a text file (*cat* is the Mac OS X command used to edit files). In this example, the full access path of the command is passed:

```
C_TEXT(input;output)
input:=""
LAUNCH EXTERNAL PROCESS("/bin/cat /folder/myfile.txt";input;output)
```

3. To get the contents of the "Users" folder (*ls -l* is the Mac OS X equivalent of the *dir* command in DOS):

```
C_TEXT($In;$Out)
LAUNCH EXTERNAL PROCESS("/bin/ls -l /Users";$In;$Out)
```

4. To launch an independent "graphic" application, it is preferable to use the *open* system command (in this case, the **LAUNCH EXTERNAL PROCESS** statement has the same effect as double-clicking the application):

```
LAUNCH EXTERNAL PROCESS("open /Applications/Calculator.app")
```

## Examples under Windows

5. To open NotePad:

```
LAUNCH EXTERNAL PROCESS("C:¥¥WINDOWS¥¥notepad.exe")
```

6. To open Notepad and open a specific document:

```
LAUNCH EXTERNAL PROCESS("C:¥¥WINDOWS¥¥notepad.exe C:¥¥Docs¥¥new folder¥¥res.txt")
```

7. To launch the Microsoft® Word® application and open a specific document (note the use of the two ""):

```
$mydoc:="C:¥¥Program Files¥¥Microsoft Office¥¥Office10¥¥WINWORD.EXE ¥"C:¥¥Documents and
Settings¥¥Mark¥¥Desktop¥¥MyDocs¥¥New folder¥¥test.xml¥""
LAUNCH EXTERNAL PROCESS($mydoc;$tIn;$tOut)
```

8. To execute a Perl script (requires ActivePerl):

```
C_TEXT($input;$output)
SET ENVIRONMENT VARIABLE("myvariable";"value")
LAUNCH EXTERNAL PROCESS("D:¥¥Perl¥¥bin¥¥perl.exe D:¥¥Perl¥¥eg¥¥cgi¥¥env.pl";$input;$output)
```

9. To launch a command with the current directory and without displaying the console:

```
SET ENVIRONMENT VARIABLE("_4D_OPTION_CURRENT_DIRECTORY";"C:¥¥4D_VCS")
SET ENVIRONMENT VARIABLE("_4D_OPTION_HIDE_CONSOLE";"true")
LAUNCH EXTERNAL PROCESS("mycommand")
```

10. To allow the user to open an external document on Windows:

```
$docname:=Select document("";"*.*";"Choose the file to open";0)
If(OK=1)
   SET ENVIRONMENT VARIABLE("_4D_OPTION_HIDE_CONSOLE";"true")
   LAUNCH EXTERNAL PROCESS("cmd.exe /C start ¥"¥" ¥""+$docname+"¥"")
End if
```

11. The following examples request the process list on Windows:

```
C_LONGINT($pid)
C_TEXT($stdin;$stdout;$stderr)

LAUNCH EXTERNAL PROCESS("tasklist";$pid) //gets PID only
LAUNCH EXTERNAL PROCESS("tasklist";$stdin;$stdout;$stderr;$pid) //gets all information
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise (file not found, insufficient memory, etc.), it is set to 0.

# OPEN URL

OPEN URL ( path {; appName}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| path | String | ⇒ | Path of document or URL to open |
| appName | String | ⇒ | Name of application to use |
| * | Operator | ⇒ | If specified = URL is not translated, If omitted = URL is translated |

## Description

The **OPEN URL** command opens the file or URL passed in the *path* parameter with the application indicated in *appName* (if any).

The *path* parameter can contain either a standard URL or a file pathname. The command accepts colons (':') under OS X, slashes ('¥') under Windows or a Posix URL beginning with file://.

If the *appName* parameter is omitted, 4D first attempts to interpret the *path* parameter as a file pathname. If this is the case, 4D will request the system to open the file using the most suitable application (for example, a browser for .html files, Word for .doc files, etc.). The * parameter is ignored in this case.
If the *path* parameter contains a standard URL (mailto:, news:, http:, etc. protocols), 4D starts the default Web browser and accesses the URL. If there is no browser on the volumes connected to the computer, this command has no effect.

When the *appName* parameter is passed, the command interrogates the system. If an application with this name is installed, it is started and the command requests it to open the specified URL or document.
Under Windows, the mechanism for recognizing the application name is the same as the one used by the "Run" command of the Start menu. For example, you could pass:

- "iexplore" to start the Internet Explorer.
- "chrome" to start Chrome (if installed)
- "winword" to start MS Word (if installed)

**Note:** You will find the list of applications installed in the *registry* at the following key:
HKEY_LOCAL_MACHINE¥SOFTWARE¥Microsoft¥Windows¥CurrentVersion¥App Paths

Under OS X, the mechanism uses the Finder which automatically indexes all the applications installed. It can recognize any .app application by means of its package name (with or without the .app suffix). For example, you could pass:

- "safari"
- "FireFox"
- "TextEdit"

When the *appName* application is not found, no error is returned; the command is run as if this parameter had not been specified.

4D automatically encodes the URL's special characters. If you pass the * character, 4D will not translate the URL's special characters. This option allows you to access and to send URLs of the type: "*http://www.server.net/page.htm?q=something*".

**Note:** This command does not work when called from a Web process.

## Example 1

The following examples illustrate different types of strings that are accepted as URLs by the command:

```
OPEN URL("http://www.4d.com")
OPEN URL("file://C:/Users/Laurent/Documents/pending.htm")
OPEN URL("C:¥¥Users¥¥Laurent¥¥Documents¥¥pending.htm")
OPEN URL("mailto:jean_martin@4d.fr")
```

## Example 2

This example can be used to launch the most suitable application:

```
$file:=Select document("";"";0)
If(OK=1)
   OPEN URL(Document)
End if
```

## Example 3

The *appName* parameter lets you open the same text file using different applications:

```
OPEN URL("C:¥¥temp¥¥cookies.txt") //open the file with Notepad
OPEN URL("C:¥¥temp¥¥cookies.txt";"winword") //open the file with MS Word (if installed)
OPEN URL("C:¥¥temp¥¥cookies.txt";"excel") //open the file with MS Excel (if installed)
```

# PROCESS 4D TAGS

PROCESS 4D TAGS ( inputTemplate ; outputResult {; param}{; param2 ; ... ; paramN} )

| Parameter | Type | | Description |
|---|---|---|---|
| inputTemplate | Text, BLOB | ➡ | Data containing tags to process |
| outputResult | Text, BLOB | ⬅ | Result from template execution |
| param | Text, Number, Date, Time, Pointer | ➡ | Parameter(s) passed to template being executed |

## Description

The **PROCESS 4D TAGS** command causes the processing of 4D transformation tags contained in the *inputTemplate* parameter (field or variable of the BLOB or Text type) while (optionally) inserting value(s) using the *param* parameters and returns the result in *outputResult*. For a complete description of these tags, refer to the **4D Transformation Tags** section.

This command lets you execute a "template" type text containing tags and references to 4D expressions and/or variables, and to produce a result depending on the execution context and/or the values passed as parameters.
For example, you can use this command to generate and save HTML pages based on **semi-dynamic pages** containing 4D transformation tags (without it being necessary for 4D's Web server to be started). You can use it to send e-mail messages in HTML format that contain processing of and/or references to data contained in the database via the 4D Internet Commands. It is possible to process any type of data based on text, such as XML, SVG or multi-style text.

Pass the data containing the tags to be processed in the *inputTemplate* parameter. This parameter can be a field or variable of the BLOB or Text type. The Text type is usually sufficient (parameters can receive up to 2 GB of text).

**Compatibility note**: Beginning with version 12 of 4D, when you use BLOB type parameters, the command automatically considers that the character set used for BLOBs is MacRoman. For better efficiency, it is strongly recommended to use Text type parameters for which processing is carried out in Unicode mode.

All the transformation tags of 4D are supported (*4DTEXT*, *4DHTML, 4DSCRIPT*, *4DLOOP*, *4DEVAL*, etc.).

**Note:** When using the *4DINCLUDE* tag outside the framework of the Web server (Web process):

- with 4D in local mode or 4D Server, the default folder is the folder containing the database structure file,
- with 4D in remote mode, the default folder is the folder containing the 4D application.

The **PROCESS 4D TAGS** command supports an indefinite number of *param* parameters that can be inserted into the executed code. As with project methods, these parameters can contain scalar values of varied types (text, date, time, longint, real, etc.). You can also use arrays, by means of array pointers. Inside the code processed by the 4D tags, these parameters can be accessed by means of standard arguments ($1, $2, etc.), just like in 4D methods 4D (see example).
A dedicated set of local variables is defined in the execution context of the **PROCESS 4D TAGS** command. These variables can be written or read during processing.

**Compatibility note:** In previous versions of 4D, local variables defined in the calling context could be accessed in the **PROCESS 4D TAGS** execution context in interpreted mode. Beginning with 4D v14 R4, this is not the case anymore.

After command execution, the *outputResult* parameter receives the execution result of the *inputTemplate* parameter, along with the result of the processing of any 4D tags that it contains, when applicable. If *inputTemplate* does not contain any 4D tags, the contents of *outputResult* is identical to that of *inputTemplate*.
The *outputResult* parameter may be a field or a variable, but it must be of the same type as that of the *inputTemplate* parameter.

**Note:** This command never calls the **On Web Authentication database method**.

## Example 1

This example loads a 'template' type document, processes the tags it contains and then stores it:

```
C_BLOB($Blob_x)
C_BLOB($blob_out)
C_TEXT($inputText_t)
C_TEXT($outputText_t)

DOCUMENT TO BLOB("mytemplate.txt";$Blob_x)
$inputText_t:=BLOB to text($Blob_x;UTF8 text without length)
```

```
PROCESS 4D TAGS($inputText_t;$outputText_t)
TEXT TO BLOB($outputText_t;$blob_out;UTF8 text without length)
BLOB TO DOCUMENT($document;$blob_out)
```

## Example 2

This example generates a text using data of the arrays:

```
ARRAY TEXT($array;2)
$array{1}:="hello"
$array{2}:="world"
$input:="<!--#4DEVAL $1-->"
$input:=$input+"<!--#4DLOOP $2-->"
$input:=$input+"<!--#4DEVAL $2->{$2->}--> "
$input:=$input+"<!--#4DENDLOOP-->"
PROCESS 4D TAGS($input;$output;"elements = ";->$array)
 // $output = "elements = hello world"
```

# SET ENVIRONMENT VARIABLE

SET ENVIRONMENT VARIABLE ( varName ; varValue )

| Parameter | Type | | Description |
|---|---|---|---|
| varName | String | ⇒ | Variable name to set |
| varValue | String | ⇒ | Value of the variable or "" to reset default value |

## Description

The **SET ENVIRONMENT VARIABLE** command allows you to set the value of an environment variable under Mac OS X and Windows. It is meant to be used with the **LAUNCH EXTERNAL PROCESS** command. It also works with the **PHP Execute** command.

Pass the name of the variable to define in *varName* and its value in *varValue*.

- To get the general list of environment variables and possible values, please refer to the technical documentation of your operating system.
- To see the list of environment variables available with the **LAUNCH EXTERNAL PROCESS** command, please refer to the documentation for this command. Note that three specific environment variables are available for use in this context:
  *_4D_OPTION_CURRENT_DIRECTORY*: Used to set the current directory of the external process to be launched. In *varValue*, you must pass the pathname of the directory (HFS type syntax on Mac OS and DOS on Windows).
  *_4D_OPTION_HIDE_CONSOLE* (Windows only): Used to hide the window of the DOS console. You must pass "true" in *varValue* to hide the console or "false" to display it.
  *_4D_OPTION_BLOCKING_EXTERNAL_PROCESS*: Used to execute the external process in asynchronous mode, in other words, non-blocking for other applications. You must pass "false" in *varValue* to set an asynchronous execution or "true" to set a synchronous execution (no default value can be set for this variable).
  These variables are valid in the current process for the next call to **LAUNCH EXTERNAL PROCESS**.

## Example

Refer to examples of the **LAUNCH EXTERNAL PROCESS** command.

# SET MACRO PARAMETER

SET MACRO PARAMETER ( selector ; textParam )

| Parameter | Type | | Description |
|---|---|---|---|
| selector | Longint | ➡ | Selection to use |
| textParam | Text | ➡ | Text sent |

## Description

The **SET MACRO PARAMETER** command inserts the *paramText* text into the method from which it has been called.

If text has been selected in the method, the *selector* parameter can be used to set whether the *paramText* text must replace all of the method text or only the selected text. In selector, you can pass one of the following constants, added to the "**4D Environment**" theme:

| Constant | Type | Value |
|---|---|---|
| Full method text | Longint | 1 |
| Highlighted method text | Longint | 2 |

If no text has been selected, *paramText* is inserted into the method.

## Note

In order for the **GET MACRO PARAMETER** and **SET MACRO PARAMETER** commands to work correctly, the new "version" attribute must be declared in the macro itself. The "version" attribute must be declared as follows:

```
<macro name="MyMacro" version="2">
--- Text of macro ---
</macro>
```

## Example

This macro builds a new text that will be returned to the calling method:

```
C_TEXT($input_text)
C_TEXT($output_text)
GET MACRO PARAMETER(Highlighted method text;$input_text)
 `Suppose that the selected text is a table, i.e.  "[Customers]"
$output_text:=""
$output_text:=$output_text+Command name(47)+"("+$input_text+")" ` Select all ([Customers])
$output_text:=$output_text+"$i:="+Command name(76)+"("+$input_text+")" ` $i:=Records in selection([Customers])
SET MACRO PARAMETER(Highlighted method text;$output_text)
 `Replaces the selected text by the new code
```

# Transactions

- Using Transactions
- Suspending transactions
- Active transaction
- CANCEL TRANSACTION
- In transaction
- RESUME TRANSACTION
- START TRANSACTION
- SUSPEND TRANSACTION
- Transaction level
- VALIDATE TRANSACTION

# Using Transactions

Transactions are a series of related data modifications made to a database within a process. A transaction is not saved to a database permanently until the transaction is validated. If a transaction is not completed, either because it is canceled or because of some outside event, the modifications are not saved.

During a transaction, all changes made to the database data within a process are stored locally in a temporary buffer. If the transaction is accepted with **VALIDATE TRANSACTION**, the changes are saved permanently. If the transaction is canceled with **CANCEL TRANSACTION**, the changes are not saved. In all cases, neither the current selection nor the current record are modified by the transaction management commands.

4D supports nested transactions, i.e. transactions on several hierarchical levels. The number of subtransactions allowed is unlimited. The **Transaction level** command can be used to find out the current transaction level where the code is executed.

When you use nested transactions, the result of each subtransaction depends on the validation or cancellation of the higher-level transaction. If the higher-level transaction is validated, the results of the subtransactions are confirmed (validation or cancellation). On the other hand, if the higher-level transaction is cancelled, all the subtransactions are cancelled, regardless of their respective results.

**Note:** For compatibility, nested transactions are disabled by default in databases converted from versions prior to v11 (see **Compatibility page**).

4D includes a feature allowing you to suspend and resume transactions within your 4D code. When a transaction is *suspended*, you can execute operations independently from the transaction itself and then *resume* the transaction to validate or cancel it as usual. When the transaction is suspended, you can, more particularly:

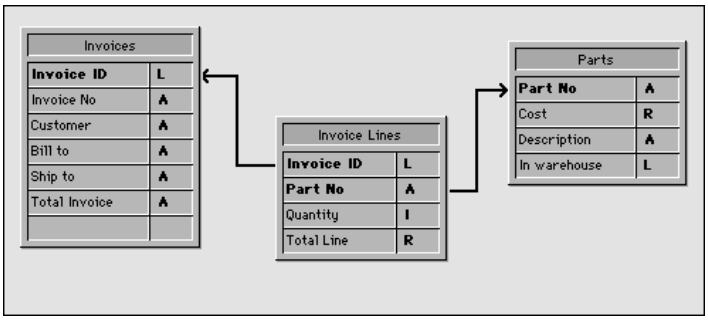
- create or modify records outside of the transaction (for example to increment an invoice number counter),
- start, suspend and/or close other transactions.

For more information about this point, refer to **Suspending transactions**.

## Transaction Examples

In this example, the database is a simple invoicing system. The invoice lines are stored in a table called [Invoice Lines], which is related to the table [Invoices] by means of a relation between the fields [Invoices]Invoice ID and [Invoice Lines]Invoice ID. When an invoice is added, a unique ID is calculated, using the **Sequence number** command. The relation between [Invoices] and [Invoice Lines] is an automatic Relate Many relation. The Auto assign related value in subform check box is checked.

The relation between [Invoice Lines] and [Parts] is manual.




When a user enters an invoice, the following actions are executed:

- Add a record in the table [Invoices].
- Add several records in the table [Invoice Lines].
- Update the [Parts]In Warehouse field of each part listed in the invoice.

This example is a typical situation in which you need to use a transaction. You must be sure that you can save all these records during the operation or that you will be able to cancel the transaction if a record cannot be added or updated. In other words, you must save related data.

If you do not use a transaction, you cannot guarantee the logical data integrity of your database. For example, if one record of the [Parts] records is locked, you will not be able to update the quantity stored in the field [Parts]In Warehouse.

Therefore, this field will become logically incorrect. The sum of the parts sold and the parts remaining in the warehouse will not be equal to the original quantity entered in the record. You can avoid such a situation by using transactions.

There are several ways of performing data entry using transactions:

1. You can handle the transactions yourself by using the transaction commands **START TRANSACTION**, **VALIDATE TRANSACTION** and **CANCEL TRANSACTION**. You can write, for example:

```
READ WRITE([Invoice Lines])
READ WRITE([Parts])
FORM SET INPUT([Invoices];"Input")
Repeat
   START TRANSACTION
   ADD RECORD([Invoices])
   If(OK=1)
      VALIDATE TRANSACTION
   Else
      CANCEL TRANSACTION
   End if
Until(OK=0)
READ ONLY(*)
```

2. To reduce record locking while performing the data entry, you can also choose to manage transactions from within the form method and access the tables in **READ WRITE** only when it becomes necessary.

You perform the data entry using the input form for [Invoices], which contains the related table [Invoice Lines] in a subform. The form has two buttons: *bCancel* and *bOK*, both of which are no action buttons.
The adding loop becomes:

```
READ WRITE([Invoice Lines])
READ ONLY([Parts])
FORM SET INPUT([Invoices];"Input")
Repeat
   ADD RECORD([Invoices])
Until(bOK=0)
READ ONLY([Invoice Lines])
```

Note that the [Parts] table is now in read-only access mode during data entry. Read/write access will be available only if the data entry is validated.

The transaction is started in the [Invoices] input form method listed here:

```
Case of
   :(Form event=On Load)
      START TRANSACTION
      [Invoices]Invoice ID:=Sequence number([Invoices]Invoice ID)
   Else
      [Invoices]Total Invoice:=Sum([Invoice Lines]Total line)
End case
```

If you click the *bCancel* button, the data entry as well as the transaction must be canceled.
Here is the object method of the *bCancel* button:

```
Case of
   :(Form event=On Clicked)
      CANCEL TRANSACTION
      CANCEL
End case
```

If you click the *bValidate* button, the data entry must be accepted and the transaction must be validated. Here is the object method of the *bOK* button:

```
Case of
   :(Form event=On Clicked)
      $NbLines:=Records in selection([Invoice Lines])
      READ WRITE([Parts]) ` Switch to Read/Write access for the [Parts] table
      FIRST RECORD([Invoice Lines]) ` Start at the first line
      $ValidTrans:=True ` Assume everything will be OK
      For($Line;1;$NbLines) ` For each line
         RELATE ONE([Invoice Lines]Part No)
```

```
            OK:=1 ` Assume you want to continue
            While(Locked([Parts]) & (OK=1)) ` Try getting the record in Read/Write access
               CONFIRM("The Part "+[Invoice Lines]Part No+" is in use. Wait?")
               If(OK=1)
                  DELAY PROCESS(Current process;60)
                  LOAD RECORD([Parts])
               End if
            End while
            If(OK=1)
   ` Update quantity in the warehouse
               [Parts]In Warehouse:=[Parts]In Warehouse-[Invoice Lines]Quantity
               SAVE RECORD([Parts]) ` Save the record
            Else
               $Line:=$NbLines+1 ` Leave the loop
               $ValidTrans:=False
            End if
            NEXT RECORD([Invoice Lines]) ` Go next line
         End for
         READ ONLY([Parts]) ` Set the table state to read only
         If($ValidTrans)
            SAVE RECORD([Invoices]) ` Save the Invoices record
            VALIDATE TRANSACTION ` Validate all database modifications
         Else
            CANCEL TRANSACTION ` Cancel everything
         End if
         CANCEL ` Leave the form
   End case
```

In this code, we call the **CANCEL** command regardless of the button clicked. The new record is not validated by a call to **ACCEPT**, but by the **SAVE RECORD** command. In addition, note that **SAVE RECORD** is called just before the **VALIDATE TRANSACTION** command. Therefore, saving the [Invoices] record is actually a part of the transaction. Calling the **ACCEPT** command would also validate the record, but in this case the transaction would be validated before the [Invoices] record was saved. In other words, the record would be saved outside the transaction.

Depending on your needs, you can customize your database, as shown in these examples. In the last example, the handling of locked records in the [Parts] table could be developed further.

# Suspending transactions

## Principle

Suspending a transaction is useful when you need to perform, from within a transaction, certain operations that do not need to be executed under the control of this transaction. For example, imagine the case where a customer places an order, thus within a transaction, and also updates their address. Next the customer changes their mind and cancels the order. The transaction is cancelled, but you do not want the address change to be reverted. This is a typical example where suspending the transaction is useful. Three commands are used to suspend and resume transactions:

- **SUSPEND TRANSACTION**: pauses current transaction. Any updated or added records remain locked.
- **RESUME TRANSACTION**: reactivates a suspended transaction.
- **Active transaction**: returns **False** if the transaction is suspended or if there is no current transaction, and **True** if it is started or resumed.

## Example

This example illustrates the need for a suspended transaction. In an Invoices database, we want to get a new invoice number during a transaction. This number is computed and stored in a [Settings] table. In a multi-user environment, concurrent accesses must be protected; however, because of the transaction, the [Settings] table could be locked by another user even though this data is independent from the main transaction. In this case, you can suspend the transaction when accessing the table.

```
 //Standard method that creates an invoice
START TRANSACTION
...
CREATE RECORD([Invoices])
[Invoices]InvoiceID:=GetInvoiceNum //call the method to get an available number
...
SAVE RECORD([Invoices])
VALIDATE TRANSACTION
```

The ***GetInvoiceNum*** method suspends the transaction before executing. Note that this code will work even when the method is called from outside of a transaction:

```
 //GetInvoiceNum project method
 //GetInvoiceNum -> Next available invoice number
C_LONGINT($0)
SUSPEND TRANSACTION
ALL RECORDS([Settings])
If(Locked([Settings])) //multi-user access
   While(Locked([Settings]))
      MESSAGE("Waiting for locked Settings record")
      DELAY PROCESS(Current process;30)
      LOAD RECORD([Settings])
   End while
End if
[Settings]InvoiceNum:=[Settings]InvoiceNum+1
$0:=[Settings]InvoiceNum
SAVE RECORD([Settings])
UNLOAD RECORD([Settings])
RESUME TRANSACTION
```

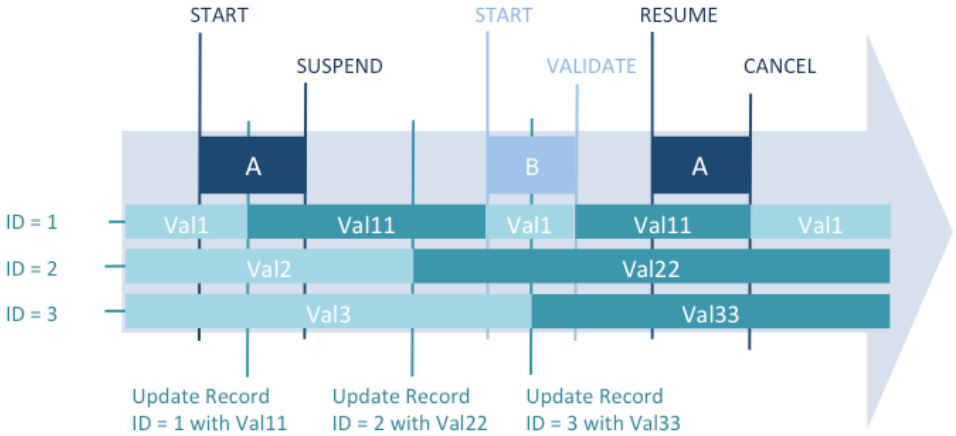
## Detailed operation

### How does a suspended transaction work?

When a transaction is suspended, the following principles are implemented:

- You can access records that were added or modified during the transaction, and you cannot see any records that were deleted during the transaction.
- You can create, save, delete, or modify records outside the transaction.
- You can start a new transaction, but within this included transaction you will not be able to see any records or record values that were added or modified during the suspended transaction. In fact, this new transaction is totally independent from the suspended one, similar to a transaction of another process, and since the suspended transaction could later be resumed or canceled, any added or modified records are automatically hidden for the new transaction. As soon as you commit or cancel the new transaction, you can see these records again.
- Any records that are modified, deleted or added within the suspended transaction remain locked for other processes. If you try to modify or delete these records outside the transaction or in a new transaction, an error is generated.

These implementations are summarized in the following graphic:




*Values edited during transaction A (ID1 record gets Val11) are not available in a new transaction (B) created during the "suspended" period. Values edited during the "suspended" period (ID2 record gets Val22 and ID3 record gets Val33) are saved even after transaction A is cancelled.*

Specific features have been added to handle errors:

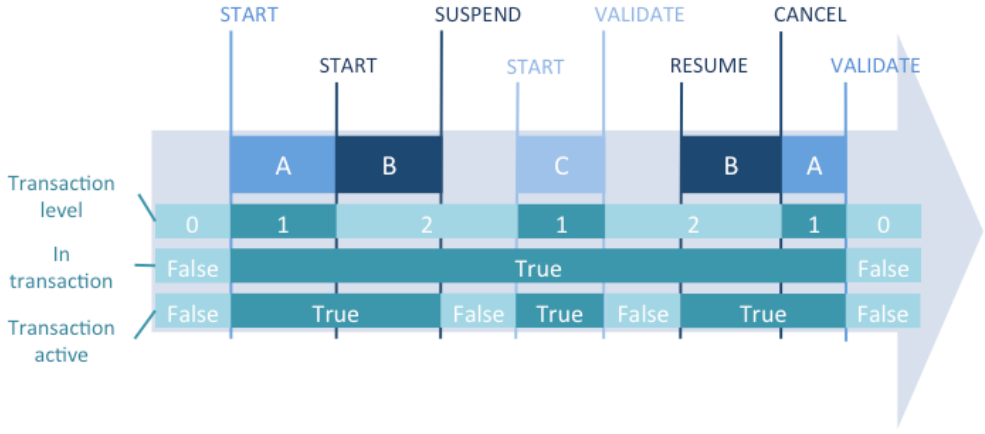
- The current record of each table becomes temporarily locked if it is modified during the transaction and is automatically unlocked when the transaction is resumed. This mechanism is important to prevent unwanted saves on parts of the transaction.
- If you execute an invalid sequence such as *start transaction / suspend transaction / start transaction / resume transaction*, an error is generated. This mechanism prevents developers from forgetting to commit or cancel any included transactions before resuming the suspended transaction.

### Suspended transactions and process status

The existing **In transaction** command returns **True** when a transaction has been started, even if it is suspended. To find out whether the current transaction is suspended, you need to use the new **Transaction active** command, which returns **False** in this case.

Both commands, however, also return **False** if no transaction has been started. You may then need to use the existing **Transaction level** command, which returns 0 in this context (no transaction started).

The following graphic illustrates the various transaction contexts and the corresponding values returned by the transaction commands:



# Active transaction

Active transaction -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Boolean | Returns False if the current transaction is suspended |

## Description

The **Active transaction** command returns **True** if the current process is in transaction, and this transaction is not suspended. It returns **False** if there is no current transaction, or if the current transaction is suspended. A transaction can be suspended using the **SUSPEND TRANSACTION** command.

Since the command will also return **False** if the current process is not in transaction, you may need to check using the **In transaction** command to know whether the process is in transaction.

For more information, please refer to **Suspending transactions**.

## Description

You want to know the current transaction status:

```
If(In transaction)
   If(Not(Active transaction))
      ALERT("The current transaction is suspended")
   Else
      ALERT("The current transaction is active")
   End if
Else
   ALERT("We are not in transaction")
End if
```

# CANCEL TRANSACTION

CANCEL TRANSACTION

Does not require any parameters

## Description

**CANCEL TRANSACTION** cancels the transaction that was started with **START TRANSACTION** of the corresponding level in the current process. **CANCEL TRANSACTION** cancels the operations executed on the data and stored during the transaction.

**Note: CANCEL TRANSACTION** does not have an effect on any changes made in the current records that were not saved - they remain displayed after the command is executed.

# In transaction

In transaction -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | Returns TRUE if current process is in transaction |

## Description

The **In transaction** command returns **TRUE** if the current process is in a transaction, otherwise it returns **FALSE**.

## Example

If you perform a multi-record operation (adding, modifying, or deleting records), you may encounter locked records. In this case, if you have to maintain data integrity, you must be in transaction so you can “roll-back” the whole operation and leave the database untouched.

If you perform the operation from within a trigger or from a subroutine (that can be called while in transaction or not), you can use **In transaction** to check whether or not the current process method or the caller method started a transaction. If a transaction was not started, you do not even start the operation, because you already know that you will not be able to roll it back if it fails.

# RESUME TRANSACTION

RESUME TRANSACTION
Does not require any parameters

## Description

The **RESUME TRANSACTION** command resumes the transaction that was paused using **SUSPEND TRANSACTION** at the corresponding level in the current process. Any operations that are executed after this command are carried out under transaction control (except when several suspended transactions are nested).

For more information, please refer to **Suspending transactions**.

# START TRANSACTION

START TRANSACTION
Does not require any parameters

## Description

**START TRANSACTION** starts a transaction in the current process. All changes to the data (records) of the database within the transaction are stored temporarily until the transaction is accepted (validated) or canceled.

Beginning with version 11 of 4D, you can nest several transactions (sub-transactions). Each transaction or sub-transaction must eventually be cancelled or validated. Note that if the main transaction is cancelled, all the sub-transactions are cancelled as well, regardless of their result.

# SUSPEND TRANSACTION

SUSPEND TRANSACTION
Does not require any parameters

## Description

The **SUSPEND TRANSACTION** command pauses the current transaction in the current process. You can then handle data in other parts of the database, for example, without it being included in the transaction, and while preserving the transaction context untouched. Any records that have been updated or added in the transaction are locked until the transaction is resumed using the **RESUME TRANSACTION** command.

For more information, please refer to the **Suspending transactions** section.

# Transaction level

Transaction level -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Current transaction level (0 if no transaction has been started) |

## Description

The **Transaction level** command returns the current transaction level for the process. This command takes all the transactions of the current process into account, regardless of whether they were started via the 4D language or via SQL.

# VALIDATE TRANSACTION

VALIDATE TRANSACTION
Does not require any parameters

## Description

**VALIDATE TRANSACTION** accepts the transaction that was started with **START TRANSACTION** of the corresponding level in the current process. The command saves the changes to the data of the database that occurred during the transaction.

Starting with version 11 of 4D, you can nest several transactions (sub-transactions). If the main transaction is cancelled, all the sub-transactions are cancelled, even if they have been validated individually using this command.

## System variables and sets

The system variable OK is set to 1 if the transaction has been validated correctly; otherwise, it is set to 0.

Note that when OK is set to 0, the transaction is automatically cancelled internally (equivalent to **CANCEL TRANSACTION**). Consequently, you must not explicitly call **CANCEL TRANSACTION** if OK=0, particularly in the context of nested transactions, because the cancellation will then be applied to the higher level transaction.

# Triggers

- Triggers
- Trigger event
- Trigger level
- TRIGGER PROPERTIES

# Triggers

A Trigger is a method attached to a table. It is a property of a table. You do not call triggers; they are automatically invoked by the 4D database engine each time you manipulate table records (add, delete and modify). You can write very simple triggers, and then make them more sophisticated.

Triggers can prevent “illegal” operations on the records of your database. They are a very powerful tool for restricting operations on a table, as well as preventing accidental data loss or tampering. For example, in an invoicing system, you can prevent anyone from adding an invoice without specifying the customer to whom the invoice is billed.

## Activating and Creating a Trigger

By default, when you create a table in the Design Environment, it has no trigger.

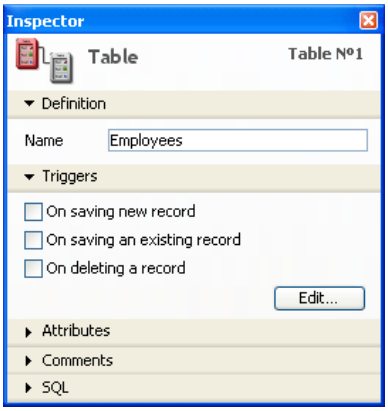
To use a trigger for a table, you need to:

- Activate the trigger and tell 4D when it has to be invoked.
- Write the code for the trigger.

Activating a trigger that is not yet written or writing a trigger without activating it will not affect the operations performed on a table.

### 1. Activating a Trigger

To activate a trigger for a table, you must select one of the **Triggers** options (database events) for the table in the Inspector window of the structure:




***On saving an existing record***

If this option is selected, the trigger will be invoked each time a record of the table is modified.
This happens when:

- Modifying a record in data entry (Design environment, **MODIFY RECORD** command or the SQL **UPDATE** command).
- Saving an already existing record using **SAVE RECORD**.
- Calling any other commands that save existing records (i.e., **ARRAY TO SELECTION**, **APPLY TO SELECTION**, etc.).
- Using a Plug-in that calls the **SAVE RECORD** command.

**Note:** For optimization reasons, the trigger is not called when the record is saved by the user or via the **SAVE RECORD** command if no field in the table has been modified in the record. If you want to "force" the calling of the trigger in this case, you can simply assign a field to itself:

```
[thetable]thefield:=[thetable]thefield
```

***On deleting a record***

If this option is selected, the trigger will be invoked each time a record of the table is deleted.
This happens when:

- Deleting a record (Design environment or calling **DELETE RECORD**, **DELETE SELECTION** or the SQL **DELETE** command).
- Performing any operation that provokes deletion of related records through the deletion control options of a relation.
- Using a Plug-in that calls the **DELETE RECORD** command.

**Note:** The **TRUNCATE TABLE** command does NOT call the trigger.

***On saving a new record***

If this option is selected, the trigger will be invoked each time a record is added to the table.
This happens when:

- Adding a record in data entry (Design environment, **ADD RECORD** command or the SQL **INSERT** command).
- Creating and saving a record with **CREATE RECORD** and **SAVE RECORD**. Note that the trigger is invoked at the moment you call **SAVE RECORD**, not when it is created.
- Importing records (Design environment or using an import command).
- Calling any other commands that create and/or save new records (i.e., **ARRAY TO SELECTION**, **SAVE RELATED ONE**, etc.).
- Using a Plug-in that calls the **CREATE RECORD** and **SAVE RECORD** commands.

### 2. Creating a Trigger

To create a trigger for a table, use the **Explorer** Window, click on the **Edit...** button in the Inspector window of the structure, or press Alt (on Windows) or Option (Macintosh) and double-click on the table title in the Structure window. For more information, see the 4D Design Reference manual.

## Database Events

A trigger can be invoked for one of the three **database events** described above. Within the trigger, you detect which event is occurring by calling the **Trigger event** function. This function returns a numeric value that denotes the database event.

Typically, you write a trigger with a **Case of** structure on the result returned by **Trigger event**. You can use the constants of the **_o_LAST SUBRECORD** theme:

```
 //  Trigger for [anyTable]
C_LONGINT($0)
$0:=0 // Assume the database request will be granted
Case of
   :(Trigger event=On Saving New Record Event)
 // Perform appropriate actions for the saving of a newly created record
   :(Trigger event=On Saving Existing Record Event)
 // Perform appropriate actions for the saving of an already existing record
   :(Trigger event=On Deleting Record Event)
 // Perform appropriate actions for the deletion of a record
End case
```

## Triggers are Functions

A trigger has two purposes:

- Performing actions on the record just before it is saved or deleted.
- Granting or rejecting a database operation.

### 1. Performing Actions

Each time a record is saved (added or modified) to a *[Documents]* table, you want to “mark” the record with a time stamp for creation and another one for the most recent modification. You can write the following trigger:

```
 // Trigger for table [Documents]
Case of
   :(Trigger event=On Saving New Record Event)
      [Documents]Creation Stamp:=Time stamp
      [Documents]Modification Stamp:=Time stamp
   :(Trigger event=On Saving Existing Record Event)
      [Documents]Modification Stamp:=Time stamp
End case
```

**Note**: The *Time stamp* function used in this example is a small project method that returns the number of seconds elapsed since a fixed date was chosen arbitrarily.

After this trigger has been written and activated, no matter what way you add or modify a record to the *[Documents]* table (data entry, import, project method, 4D plug-in), the fields *[Documents]Creation Stamp* and *[Documents]Modification Stamp* will automatically be assigned by the trigger before the record is eventually written to the disk.

**Note**: See the example for the **GET DOCUMENT PROPERTIES** command for a complete study of this example.

### 2. Granting or rejecting the database operation

To grant or reject a database operation, the trigger must return a **trigger error code** in the *$0* function result.

### Example

Let's take the case of an *[Employees]* table. During data entry, you enforce a rule on the field *[Employees]Social Security Number*. When you click the validation button, you check the field using the object method of the button:

```
 // bAccept button object method
If(Good SS number([Employees]SS number))
   ACCEPT
Else
   BEEP
   ALERT("Enter a Social Security Number then click OK again.")
End if
```

If the field value is valid, you accept the data entry; if the field value is not valid, you display an alert and you stay in data entry.

If you also create *[Employees]* records programmatically, the following piece of code would be programmatically valid, but would violate the rule expressed in the previous object method:

```
 // Extract from a project method
 // ...
CREATE RECORD([Employees])
[Employees]Name:="DOE"
SAVE RECORD([Employees]) // <-- DB rule violation! The SS number has not been assigned!
 // ...
```

Using a trigger for the *[Employees]*table, you can enforce the *[Employees]SS number* rule at all the levels of the database. The trigger would look like this:

```
 // Trigger for [Employees]
$0:=0
$dbEvent:=Trigger event
Case of
   :(($dbEvent=On Saving New Record Event)|($dbEvent=On Saving Existing Record Event))
      If(Not(Good SS number([Employees]SS number)))
         $0:=-15050
      Else
 // ...
      End if
 // ...
End case
```

Once this trigger is written and activated, the line **SAVE RECORD** ([Employees]) will generate a database engine error -15050, and the record will NOT be saved.

Similarly, if a 4D Plug-in attempted to save an *[Employees]* record with an invalid social security number, the trigger will generate the same error and the record will not be saved.

The trigger guarantees that nobody (user, database designer, plug-in) can violate the social security number rule, either deliberately or accidentally.

Note that even if you do not have a trigger for a table, you can get database engine errors while attempting to save or delete a record. For example, if you attempt to save a record with a duplicated value in a unique indexed field, the error -9998 is returned.

Therefore, triggers returning errors add new database engine errors to your application:

- 4D manages the "regular" errors: unique index, relational data control, and so on.
- Using triggers, you manage the custom errors unique to your application.

**Important**: You can return an error code value of your choice. However, do NOT use error codes already taken by the 4D database engine. We strongly recommend that you use error codes between -32000 and -15000. We reserve error codes above -15000 for the database engine.

At the process level, you handle trigger errors the same way you handle database engine errors:

- You can let 4D display the standard error dialog box, then the method is halted.
- You can use an error-handling method installed using **ON ERR CALL** and recover the error the appropriate way.

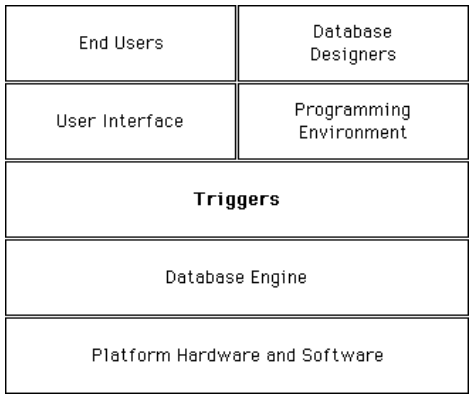
**Notes**:

- During data entry, if a trigger error is returned while attempting to validate or delete a record, the error is handled like a unique indexed error. The error dialog is displayed, and you stay in data entry. Even if you only use a database in the Design environment (not in the Application environment), you have the benefit of using triggers.
- When an error is generated by a trigger within the framework of a command acting on a selection of records (like **DELETE SELECTION**), the execution of the command is immediately stopped, without the selection having necessarily been completely processed. This case requires appropriate handling by the developer, based, for instance, on the temporary preservation of the selection, the processing and elimination of the error before trigger execution, etc.

Even when a trigger returns no error (*$0:=0*), this does not mean that a database operation will be successful—a unique index violation may occur. If the operation is the update of a record, the record may be locked, an I/O error may occur, and so on. The checking is done after the execution of the trigger. However, at the higher level of the executing process, errors returned by the database engine or a trigger are the same—a trigger error is a database engine error.

## Triggers and the 4D Architecture

Triggers execute at the database engine level. This is summarized in the following diagram:




Triggers are executed on the machine where the database engine is actually located. This is obvious with a 4D single-user version. On 4D Server, triggers are executed within the acting process on the server machine (in the "twinned" process of the process that set off the trigger), not on the client machine.

When a trigger is invoked, it executes within the context of the process that attempts the database operation. This process, which invokes the trigger execution, is called the **invoking process**.
The elements included in this context differ according to whether the database is executed with 4D in local mode or with 4D Server :

- With 4D in local mode, the trigger works with the current selections, current records, table read/write states, record locking operations, etc., of the invoking process.
- With 4D Server, only the context of the database of the invoking client process is preserved (locked records and transactional states). 4D Server also (and only) guarantees that the current record of the table of the trigger is correctly positioned. The other elements of the context (current selections for example) are those of the trigger process.

Be careful about using other database or language objects of the 4D environment, because a trigger may execute on a machine other than that of the invoking process—this is the case with 4D Server!

- **Interprocess variables**: A trigger has access to the interprocess variables of the machine where it executes. With 4D Server, it can access a machine other than that of the invoking process.
- **Process variables**: Each trigger has its own table of process variables. A trigger has no access to the process variables of the invoking process.
- **Local variables**: You can use local variables in a trigger. Their scope is the trigger execution; they are created/deleted at each execution.

- **Semaphores**: A trigger can test or set global semaphores as well as local semaphores (on the machine where it executes). However, a trigger must execute quickly, so you must be very careful when testing or setting semaphores from within triggers.
- **Sets and Named selections**: If you use a set or a named selection from within a trigger, you work on the machine where the trigger executes. In client/server mode, "process" sets and named selections (whose names do not begin with a $ nor with <>) that are created on the client machine are visible in a trigger.
- **User Interface**: Do NOT use user interface elements in a trigger (no alerts, no messages, no dialog boxes). Accordingly, you should limit any tracing of triggers in the **Debugging** window. Remember that in Client/Server, triggers execute on the 4D Server machine. An alert message on the server machine does not help a user on a client machine. Let the invoking process handle the user interface.

Note that if you use 4D's password system, you can execute the **Current user** command in the trigger in order, for example, to save the name of the user at the origin of the trigger call in a journaled table, including in client-server mode.

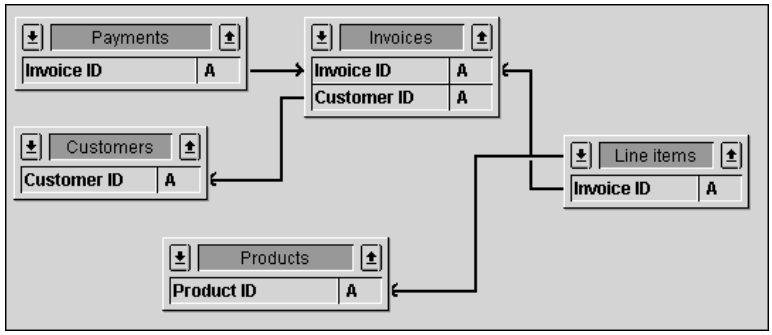
## Triggers and Transactions

Transactions must be handled at the invoking process level. They must not be managed at the trigger level. During one trigger execution, if you have to add, modify or delete multiple records (see the following case study), you must first use the **In transaction** command from within the trigger to test if the invoking process is currently in transaction. If this is not the case, the trigger may potentially encounter a locked record. Therefore, if the invoking process is not in transaction, do not even start the operations on the records. Just return an error in *$0* in order to signal to the invoking process that the database operation it is trying to perform must be executed in a transaction. Otherwise, if locked records are met, the invoking process will have no means to roll back the actions of the trigger.

**Note**: In order to optimize the combined operation of triggers and transactions, 4D does not call triggers after the execution of **VALIDATE TRANSACTION**. This prevents the triggers from being executed twice.

## Cascading Triggers

Given the following example structure:




**Note**: The tables have been collapsed; they have more fields than shown here.

Let's say that the database "authorizes" the deletion of an invoice. We can examine how such an operation would be handled at the trigger level (because you could also perform deletions at the process level).

In order to maintain the relational integrity of the data, deleting an invoice requires the following actions to be performed in the trigger for [Invoices]:

- In the [Customer] record, decrement the Gross Sales field by the amount of the invoice.
- Delete all the [Line Items] records related to the invoice.
- This also implies that the [Line Items] trigger decrements the Quantity Sold field of the [Products] record related to the line item to be deleted.
- Delete all the [Payments] records related to the deleted invoice.

First, the trigger for [Invoices] must perform these actions only if the invoking process is in transaction, so that a roll-back is possible if a locked record is met.

Second, the trigger for [Line Items] is **cascading** with the trigger for [Invoices]. The [Line Items] trigger executes "within" the execution of the [Invoices] trigger, because the deletion of the list items are consequent to a call to **DELETE SELECTION** from within the [Invoices] trigger.

Consider that all tables in this example have triggers activated for all database events. The cascade of triggers will be:

- [Invoices] trigger is invoked because the invoking process deletes an invoice
  - [Customers] trigger is invoked because the [Invoices] trigger updates the Gross Sales field
  - [Line Items] trigger is invoked because the [Invoices] trigger deletes a line item (repeated)

        - [Products] trigger is invoked because the [Line Items] trigger updates the Quantity Sold field
    - [Payments] trigger is invoked because the [Invoices] trigger deletes a payment (repeated)

In this cascade relationship, the [Invoices] trigger is said to be executing at level 1, the [Customers], [Line Items], and [Payments] triggers at level 2, and the [Products] trigger at level 3.

From within the triggers, you can use the **Trigger level** command to detect the level at which a trigger is executed. In addition, you can use the **TRIGGER PROPERTIES** command to get information about the other levels.

For example, if a [Products] record is being deleted at a process level, the [Products] trigger would be executed at level 1, not at level 3.

Using **Trigger level** and **TRIGGER PROPERTIES**, you can detect the cause of an action. In our example, an invoice is deleted at a process level. If we delete a [Customers] record at a process level, then the [Customers] trigger should attempt to delete all the invoices related to that customer. This means that the [Invoices] trigger will be invoked as above, but for another reason. From within the [Invoices] trigger, you can detect if it executed at level 1 or 2. If it did execute at level 2, you can then check whether or not it is because the [Customers] record is deleted. If this is the case, you do not even need to bother updating the Gross Sales field.

# Trigger event

Trigger event -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Longint | 0 Outside any trigger execution cycle 1 Saving a new record 2 Saving an existing record 3 Deleting a record |

## Description

Called from within a trigger, the **Trigger event** command returns a numeric value that denotes the type of the database event, in other words, the reason why the trigger has been invoked.

The following predefined constants are provided in the **Trigger Events** theme:

| Constant | Type | Value |
|---|---|---|
| On Deleting Record Event | Longint | 3 |
| On Saving Existing Record Event | Longint | 2 |
| On Saving New Record Event | Longint | 1 |

Within a trigger, if you perform database operations on multiple records, you may encounter conditions (usually locked records) that will make the trigger unable to perform correctly. An example of this situation is updating multiple records in a [Products] table when a record is being added to an [Invoices] table. At this point, you must stop attempting database operations, and return a database error so the invoking process will know that its database request cannot be performed. Then the invoking process must be able to cancel, during the transaction, the incomplete database operations performed by the trigger. When this type of situation occurs, you need to know from within the trigger if you are in transaction even before attempting anything. To do so, use the command **In transaction**.

When cascading trigger calls, 4D has no limit other than the available memory. To optimize trigger execution, you may want to write the code of your triggers depending not only on the database event, but also on the level of the call when triggers are cascaded. For example, during a deletion database event for the [Invoices] table, you may want to skip the update of the [Customers] Gross Sales field if the deletion of the [Invoices] record is part of the deletion of **all** the invoices related to a [Customers] record being deleted. To do so, use the commands **Trigger level** and **TRIGGER PROPERTIES**.

## Example

You use the **Trigger event** command to structure your triggers as follows:

```
 // Trigger for [anyTable]
C_LONGINT($0)
$0:=0 // Assume the database request will be granted
Case of
   :(Trigger event=On Saving New Record Event)
 // Perform appropriate action for the saving of a newly created record
   :(Trigger event=On Saving Existing Record Event)
 // Perform appropriate actions for the saving of an already existing record
   :(Trigger event=On Deleting Record Event)
 // Perform appropriate actions for the deletion of a record
End case
```

# Trigger level

Trigger level -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Level of trigger execution (0 if outside any trigger execution cycle) |

## Description

The **Trigger level** command returns the execution level of the trigger.

For more information on execution levels, see the topic **Cascading Triggers**.

# TRIGGER PROPERTIES

TRIGGER PROPERTIES ( triggerLevel ; dbEvent ; tableNum ; recordNum )

| Parameter | Type | | Description |
|---|---|---|---|
| triggerLevel | Longint | ⇒ | Trigger execution cycle level |
| dbEvent | Longint | ⇐ | Database event |
| tableNum | Longint | ⇐ | Involved table number |
| recordNum | Longint | ⇐ | Involved record number |

## Description

The **TRIGGER PROPERTIES** command returns information about the trigger execution level you pass in *triggerLevel*. You use this command in conjunction with **Trigger level** to perform different actions depending on the cascading of trigger execution levels. For more information, see **Triggers**.

If you pass a non-existing trigger execution level, the command returns 0 (zero) in all parameters.

The nature of the database event for the trigger execution level is returned in *dbEvent*. The following predefined constants are provided in the **Trigger Events** theme:

| Constant | Type | Value |
|---|---|---|
| On Deleting Record Event | Longint | 3 |
| On Saving Existing Record Event | Longint | 2 |
| On Saving New Record Event | Longint | 1 |

The table number and record number for the record involved by the database event for the trigger execution level are returned in *tableNum* and *recordNum*.

# User Forms


- Overview of user forms
- CREATE USER FORM
- DELETE USER FORM
- EDIT FORM
- LIST USER FORMS

# Overview of user forms

In 4D, developers can offer users the possibility of creating or modifying customized forms. These “User forms” can then be used like any other 4D form.

## Introduction

User forms are based on standard 4D forms created by the developer in the Design environment (called “source” or “developer” forms) where the **Editable by user** property has been applied in the Form editor. A simplified Form editor (called using the **EDIT FORM** command) allows users to modify the form appearance, add graphic objects (using a library of specific objects), hide elements, etc.— the developer can control of authorized actions.

User forms can be used in two different ways:

- The user modifies the “source” form and adapts it to his or her needs using the **EDIT FORM** command. The user form is kept locally and is automatically used instead of the original form.
  This behavior responds to a developer’s need to set parameters for dialog boxes while on site; for example, to add a company logo in forms, hide unnecessary fields, etc.
- The “source” file acts as a basic template that users can freely duplicate and make as many copies as necessary using the **CREATE USER FORM** command. Each copy can be set freely (content, name, etc.) using the **EDIT FORM** command. However, the name of each user form must be unique. The **FORM SET INPUT** and **FORM SET OUTPUT** commands then let you specify the user form to be used in each process.

  This behavior lets developers build, for example, customized reports for users.

## Storing and managing user forms

User form mechanisms work with both compiled and interpreted databases, with 4D in local mode, 4D Server or 4D Desktop. In client/server mode, user modified forms are available on all machines.

4D automatically handles changes in forms. When a form is set as **Editable by user**, it is locked in the Design environment. The developer must explicitly click on the icon to unlock it in order to be able to access form objects. This operation makes related user forms obsolete, which must also be regenerated. When a “source” form is deleted, the related user forms are also deleted.

User forms are stored in a separate file with a .4DA extension, placed next to the main structure file (.4DB/.4DC). This file is called “user structure file”. The behavior of this file is transparent: 4D uses a user form when it exists (the **LIST USER FORMS** command allows finding valid user forms at any time). It is also in this file that the **FORM SET INPUT** and **FORM SET OUTPUT** commands look for the user forms. When a user form is obsolete, it is deleted and 4D uses the source form by default.

In client/server, the .4DA file is broadcast on client machines following the same rules as the main structure file.

This principle also allows keeping user forms non-obsolete in case of a structure file update by the developer.

## Error codes

Specific error codes may be returned when using user form management commands. These codes, located from -9750 to -9759, are described in the **Database Engine Errors (-10602 -> 4004)** section.

## User forms and project forms

User form mechanisms are no longer compatible with project forms. The commands of the “User Forms” theme can therefore not be used with project forms.

# CREATE USER FORM

CREATE USER FORM ( aTable ; form ; userForm )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Source form table |
| form | String | ⇒ | Source table form name |
| userForm | String | ⇒ | Name of new user form |

## Description

The **CREATE USER FORM** command duplicates the 4D table *form* whose table and name are passed as parameters and creates a new user form named *userForm*.

Once created, the *userForm* form can be modified using the **EDIT FORM** command. This command allows you to create X user forms (for example, various report forms) from a single source form.

## System variables and sets

The OK variable returns 1 if the operation is executed properly; otherwise, it returns 0.

## Error management

An error is generated if:

- *form* is already a user form,
- the name of *userForm* is the same as the name of the source form or an existing user form,
- the user cannot access the form because they do not have the proper access rights.

You can intercept these errors with the error-handling method installed by the **ON ERR CALL** command.

# DELETE USER FORM

DELETE USER FORM ( aTable ; form ; userForm )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | User form table |
| form | String | ➡ | Source table form name |
| userForm | String | ➡ | User form name |

## Description

The **DELETE USER FORM** command allows you to remove the user form set using the *aTable*, *form* and *userForm* parameters.

## System variables and sets

If the user form is properly removed, the OK variable returns 1. Otherwise, OK is set to 0.

## Error management

An error is generated if:

- the user form does not exist or *userForm* contains an empty string (-9757),
- the user does not have the proper access to remove the user form.
  You can intercept this error with the error-handling method installed by the **ON ERR CALL** command.

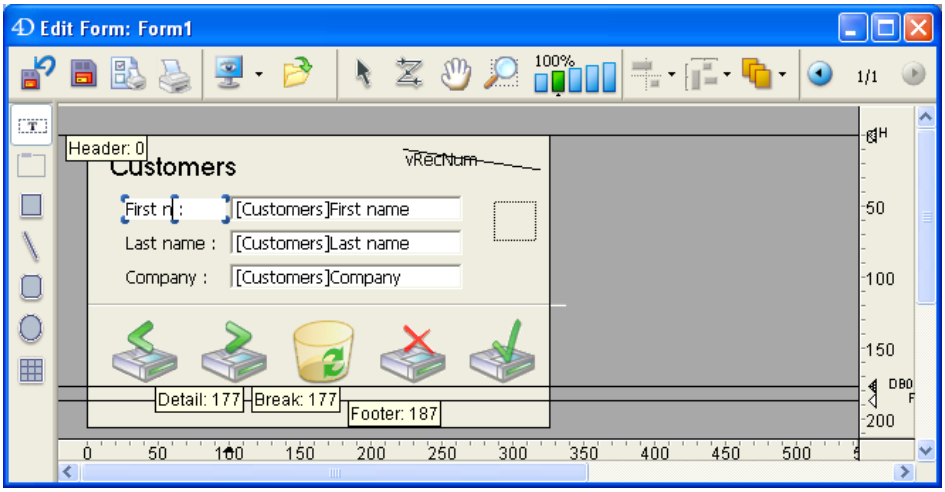
# EDIT FORM

EDIT FORM ( aTable ; form {; userForm {; library}} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table of form to modify |
| form | String | ➡ | Name of table form to modify |
| userForm | String | ➡ | Name of the user form to modify |
| library | String | ➡ | Full pathname of usable object library |

## Description

The **EDIT FORM** command opens the table form set using the a*Table*, *form* and the optional *userForm* parameters in the User form editor:




**Note:** The window of the editor opens only if it is the first window of the processs. In other words, usually you will need to open a new process to display the editor.

If you pass an empty string in the *userForm* parameter and if there is not a user form already linked to *form*, the source form is displayed in the editor. The modified form is then copied in the user structure file (.4DA) and will be used as a *form* replacement.

If a user form was already generated from *form* using this command, the user form is displayed in the editor. If you want to start from the source form, you must first delete the user form using the **DELETE USER FORM** command.

The *userForm* parameter sets a user form (created using the **CREATE USER FORM** command) to modify. In this case, the form is displayed in the editor.

In the optional *library* parameter, pass the full access path of the object library that the user will be authorized to use to customize the form. When used with the User form editor, object libraries will allow you to paste objects with their graphic properties and automatic actions. Objects with methods do not appear in the library. Be careful, it is up to the developer to make sure that the addition of library objects is compatible with the user form (and its objects) in terms of names, variables and types.

In client/server mode, the library must be found in the **Resources** folder of the database, at the same level as the **Plugins** folder, so that it is available for all client machines. If the library is valid, it is opened with the form window.

For more information on object libraries, refer to the Design Reference manual of the 4D documentation.

## Example

In this example, the user can choose a library then modify a dialog form:

```
MAP FILE TYPES("4DLB";"4IL";"4D Library")
$vALib:=Select document(1;"4DLB";"Please select a library";0)
If(OK=1)
  `A library was chosen
   $vALibPath:=Document
Else
   $vALibPath:=""
```

```
End if

EDIT FORM([Dialogs];"Welcome";"Lib_Logos.4il")
If(OK=1)
 `Display of modified form
   DIALOG([Dialogs];"Welcome")
End if
```

## System variables and sets

If the user stores the changes made to the form, the OK variable is set to 1. In case of error, OK is set to 0.

## Error management

An error is generated if:

- the form has not been declared editable by the user in the Design environment or if it does not exist,
- the form is already open and being modified in another process,
- the user cannot access the form because they do not have the proper access rights.
  You can intercept this error with the error-handling method installed by the **ON ERR CALL** command.

# LIST USER FORMS

LIST USER FORMS ( aTable ; form ; userFormArray )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇨ | Source form table |
| form | String | ⇨ | Source table form name |
| userFormArray | String array | ⇦ | Names of user forms coming from the source form |

## Description

The **LIST USER FORMS** command fills the *userFormArray* array with the names of user forms coming from the developer form (table form) set in the *table* and *form* parameters.

If the user form was created directly using the **EDIT FORM** command, the only item that *userFormArray* contains is an empty string ("").

The array is empty if there are no user forms for the specified developer form.

# User Interface

- BEEP
- Caps lock down
- Focus object
- GET FIELD TITLES
- GET MOUSE
- GET TABLE TITLES
- HIDE MENU BAR
- Macintosh command down
- Macintosh control down
- Macintosh option down
- PLAY
- Pop up menu
- POST CLICK
- POST EVENT
- POST KEY
- REDRAW
- SET ABOUT
- SET CURSOR
- SET FIELD TITLES
- SET TABLE TITLES
- Shift down
- SHOW MENU BAR
- Windows Alt down
- Windows Ctrl down
- *_o_Get platform interface*
- *_o_INVERT BACKGROUND*
- *_o_SET PLATFORM INTERFACE*

# BEEP

BEEP
Does not require any parameters

## Description

The **BEEP** command causes the PC or Macintosh to generate a beep. Your computer (on Windows or Macintosh) can emit a sound other than a beep, depending on the sound chosen in the Sound control panel.

**Warning:** Do not call **BEEP** from within a Web connection process, because the beep will be produced on the 4D Web server machine and not on the client Web browser machine.

## Example

In the following example, if no records are found by the query, a beep is emitted and an alert is displayed:

```
QUERY([Customers];[Customers]Name=$vsNameToLookFor)
If(Records in selection([Customers])=0)
   BEEP
   ALERT("There is no Customer with such a name.")
End if
```

# Caps lock down

Caps lock down -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | State of the Caps Lock key |

## Description

**Caps lock down** returns TRUE if the Caps Lock key is pressed.

## Example

See example for the **Shift down** command.

# Focus object

Focus object -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Pointer | Pointer to the object having the focus |

## Compatibility Note

This command is kept only for compatibility reasons. Starting with version 12 of 4D, it is recommended to use the **OBJECT Get pointer** command.

## Description

**Focus object** returns a pointer to the object having the focus in the current form. If no object has the focus, the command returns **Nil**. You can use **Focus object** to perform an action on a form area without having to know which object is currently selected. Be sure to test that the object is the correct data type, using **Type**, before performing an action on it.

**Note:** When it is used with a list box, the **Focus object** function returns a pointer to the list box or the column of the list box depending on the context. For more information, please refer to the **Managing List Box Objects** section.

This command cannot be used with fields in subforms.

**Note:** This command is to be used only in a data entry context; otherwise it will return errors.

## Example

The following example is an object method for a button. The object method changes the data in the current object to uppercase. The object must be a text or string data type (type 0 or 24):

```
$vp :=Focus object ` Save the pointer to the last area
Case of
   :(Nil($pointer)) ` No object has the focus
      ...
   :((Type($vp->)=Is alpha field)|(Type($vp->)=Is string var)) ` If it is a string or text area
      $vp->:=Uppercase($vp->) ` Change the area to uppercase
End case
```

# GET FIELD TITLES

GET FIELD TITLES ( aTable ; fieldTitles ; fieldNums )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇨ | Table for which you want to find out the field names |
| fieldTitles | Text array | ⇦ | Current field names |
| fieldNums | Longint array | ⇦ | Field numbers |

## Description

The **GET FIELD TITLES** command fills the *fieldTitles* and *fieldNums* arrays with the names and numbers of database fields for the desired *aTable*. The contents of these two arrays are synchronized.

If the **SET FIELD TITLES** command is called during the session, **GET FIELD TITLES** only returns the “modified” names and field numbers defined using this command.

Otherwise, **GET FIELD TITLES** returns the names of all database fields as defined in the Structure window.

In both cases, the command does not return invisible fields.

# GET MOUSE

GET MOUSE ( mouseX ; mouseY ; mouseButton {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| mouseX | Real | ⇦ | Horizontal coordinate of mouse |
| mouseY | Real | ⇦ | Vertical coordinate of mouse |
| mouseButton | Longint | ⇦ | Mouse button state: 0 = Button up 1 = Button down 2 = Right button down 3 = Both buttons down |
| * | Operator | ⇨ | If specified, global coordinate system is used If omitted, local coordinate system is used |

## Description

The **GET MOUSE** command returns the current state of the mouse.

The horizontal and vertical coordinates are returned in *mouseX* and *mouseY*. If you pass the * parameter, the coordinates are expressed relative to the screen. If you omit the * parameter, they are expressed relative to the current form window (if any) of the current process.

The parameter *mouseButton* returns the state of the buttons, as listed previously.

**Note:** The values 2 and 3 can be returned under Mac OS X starting with version 10.2.5 only.

## Example

See the example for the **Pop up menu** command.

# GET TABLE TITLES

GET TABLE TITLES ( tableTitles ; tableNums )

| Parameter | Type | | Description |
|---|---|---|---|
| tableTitles | Text array | ⇦ | Current table names |
| tableNums | Longint array | ⇦ | Table numbers |

## Description

The **GET TABLE TITLES** command fills the *tableTitles* and *tableNums* arrays with the names and numbers of database tables defined in the Structure window or using the **SET TABLE TITLES** command. The contents of these two arrays are synchronized.

If the **SET TABLE TITLES** command is called during the session, **GET TABLE TITLES** only returns the “modified” names and table numbers defined using this command.

Otherwise, **GET TABLE TITLES** returns the names of all database tables as defined in the Structure window.

In both cases, the command does not return invisible tables.

# HIDE MENU BAR

HIDE MENU BAR
Does not require any parameters

## Description

The **HIDE MENU BAR** command makes the menu bar invisible.
If the menu bar was already hidden, the command does nothing.

## Example

The following method displays a record in full-screen display (Macintosh) until you click the mouse button:

```
HIDE TOOL BAR
HIDE MENU BAR
Open window(-1;-1;1+Screen width;1+Screen height;Alternate dialog box)
FORM SET INPUT([Paintings];"Full Screen 800")
DISPLAY RECORD([Paintings])
Repeat
   GET MOUSE($vlX;$vlY;$vlButton)
Until($vlButton#0)
CLOSE WINDOW
SHOW MENU BAR
SHOW TOOL BAR
```

**Note:** On Windows, the window will be limited to the bounds of the application window.

# Macintosh command down

Macintosh command down -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | State of the Macintosh Command key (Ctrl key on Windows) |

## Description

**Macintosh command down** returns TRUE if the Macintosh command key is pressed.

**Note:** When called on a Windows platform, **Macintosh command down** returns TRUE if the Windows Ctrl key is pressed.

## Example

See example for the **Shift down** command.

# Macintosh control down

Macintosh control down -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | State of the Macintosh Control key |

## Description

**Macintosh control down** returns TRUE if the Macintosh Control key is pressed.

**Note:** When called on a Windows platform, **Macintosh control down** always return FALSE. This Macintosh key has no equivalent on Windows.

## Example

See example for the **Shift down** command.

# Macintosh option down

Macintosh option down -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | State of the Macintosh Option key (Alt key on Windows) |

## Description

**Macintosh option down** returns TRUE if the Macintosh Option key is pressed.

**Note:** When called on a Windows platform, **Macintosh option down** returns TRUE if the Windows Alt key is pressed.

## Example

See example for the **Shift down** command.

# PLAY

PLAY ( objectName {; async} )

| Parameter | Type | | Description |
|---|---|---|---|
| objectName | String | ⇒ | Name or path of sound file or system sound<br>Empty string for stopping asynchronous play |
| async | Longint | ⇒ | (Windows) If specified, asynchronous execution; If omitted, synchronous execution |

## Description

The **PLAY** command plays sound or multimedia files. You pass the full pathname of the file you want to play in *objectName*. On OS X, the command can also be used to play a system sound.

- To play a file, pass its name and pathname in *objectName*. You can pass a full pathname or a pathname relative to the database structure file.
  The main sound and multimedia file formats are supported: .WAV, .MP3, .AIFF (OS X), etc. Under OS X, the command supports more particularly the Core Audio formats.
- (OS X only) To play a system sound, pass its name directly in the *objectName* parameter.

**Note:** 'snd' resources, as used on Mac OS 9 and older, are no longer supported.

The *async* parameter specifies that the sound will play asynchronously on Windows. Synchronous play means that all processing stops until the sound has finished playing; asynchronous means that processing does not stop and the sound plays in the background. If *async* is passed and contains 0 (or any longint value), the sound is played asynchronously. If omitted, the sound is played synchronously.
**Note:** On OS X, the sound is always played asynchronously, with or without the *async* parameter.

To stop playing an asynchronous sound, use the following statement:

```
PLAY("";0)
```

## Example 1

The following example shows how to play a WAV file on Windows:

```
$DocRef :=Open document("";"WAV";Read Mode)
If(OK=1)
   CLOSE DOCUMENT($DocRef)
   PLAY(Document;0) //play asynchronously
End if
```

## Example 2

The following example code plays a system sound on OS X:

```
PLAY("Submarine.aiff")
```

# Pop up menu

Pop up menu ( contents {; default {; xCoord ; yCoord}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| contents | Text | ⇒ | Menu text definition |
| default | Longint | ⇒ | Number of menu item selected by default |
| xCoord | Longint | ⇒ | X coordinate of upper left corner |
| yCoord | Longint | ⇒ | Y coordinate of upper left corner |
| Function result | Longint | ⇐ | Selected menu item number |

## Description

The **Pop up menu** command displays a pop-up menu at the current location of the mouse.

In order to follow user interface rules, you usually call this command in response to a mouse click and if the mouse button is still down.

You define the items of the pop-up menu with the parameter *contents* as follows:

- Separate each item from the next one with a semi-colon (*;*). For example, *"ItemText1;ItemText2;ItemText3".*
- To disable an item, place an opening parenthesis (*(*) in the item text.
- To specify a separation line, pass "-" or "(-" as item text.
- To specify a font style for a line, place in the item text a less than sign (<) followed by one of these characters:

| | |
|---|---|
| <B | Bold |
| <I | Italic |
| <U | Underline |
| <O | Outline (Macintosh only) |
| <S | Shadow (Macintosh only) |

- To add a check mark to an item, place in the item text an exclamation mark (*!*) followed by the character you want as a check mark.
  - On Macintosh, the character is displayed directly. To display the standard check mark whatever the system version or language, use the following statement: *Char(18)*.
  - On Windows, a check mark is displayed, no matter what character you passed.
- To add an icon to an item, place in the item text a circumflex accent (^) followed by a character whose code plus 208 is the resource ID of a Mac OS-based icon resource.
- To add a shortcut to an item, place in the item text a slash (*/*) followed by the shortcut character for the item. Note that this last option is purely informative; no shortcut will activate the pop-up menu. However, you may want to include a shortcut if the pop-up menu item has an equivalent in the main menu bar of your application.

**Tip:** It is possible to disable the mechanism for interpreting special characters (!, /, etc.) in the pop up menu in order, for example, to have these characters included in the wording. To do this, simply have the *contents* parameter begin with the statement **Char(1)** then use this statement as a separator:

```
contents:=Char(1)+"1/4"+Char(1)+"1/2"+Char(1)+"3/4"
```

Note that once this statement has been executed, it is no longer possible to assign styles or shortcuts to the pop up menu.

The optional *default* parameter specifies the default menu item selected when the pop-up menu is displayed. Pass a value between 1 and the number of menu items. If you omit this parameter, the command selects the first menu item as the default.

The optional *xCoord* and *yCoord* parameters designate the location of the pop-up menu to be displayed. In *xCoord* and *yCoord*, pass respectively the horizontal and vertical coordinates of the upper left corner of the menu. These coordinates must be expressed in pixels in the local coordinate system of the current form. These two parameters must be passed together; if only one is passed, it will be ignored.

If you use the *xCoord* and *yCoord* parameters, the *default* parameter is ignored. In this case, the mouse is not necessarily located at the level of the pop-up menu.

These parameters are particularly useful for managing 3D buttons with an associated pop-up menu.

If you select a menu item, the command returns its number; otherwise, it returns zero (0).

**Note:** Use pop-up menus that have a reasonable number of items. If you want to display more than 50 items, you might think about a using scrollable area in a form instead of a pop-up menu.

## Example

The project method ***MY SPEED MENU*** pulls down a navigation speed menu:

```
 // MY SPEED MENU project method
GET MOUSE($vlMouseX;$vlMouseY;$vlButton)
If(Macintosh control down|($vlButton=2))
   $vtItems:="About this database...<I;(-;!-Other Options;(-"
   For($vlTable;1;Get last table number)
      If(Is table number valid($vlTable))
         $vtItems:=$vtItems+";"+Table name($vlTable)
      End if
   End for
   $vlUserChoice:=Pop up menu($vtItems)
   Case of
      :($vlUserChoice=1)
 // Display Information
      :($vlUserChoice=2)
 // Display options
      Else
         If($vlUserChoice>0)
 // Go to table whose number is $vlUserChoice-4
         End if
   End case
End if
```

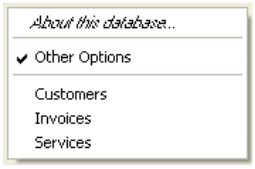
This project method can be called from:

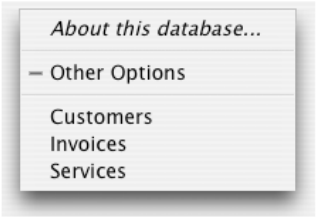
- The method of a form object that reacts to a mouse click without waiting for the mouse button to be released (i.e., an invisible button)
- A process that “spies” events and communicate with the other processes
- An event-handling method installed using **ON ERR CALL**.

In the last two cases, the click does not need to occur in any form object. This is one of the advantages of the **Pop up menu** command. Generally, you use form objects to display pop-up menus. Using **Pop up menu**, you can display the menu anywhere.

The pop-up menu is displayed on Windows by pressing the right mouse button; it is displayed on Macintosh by pressing Control-Click. Note, however, that the method does not actually check whether or not there was a mouse click; the caller method tests that.

The following is the pop-up menu as it appears on Windows (left) and Macintosh (right). Note the standard check mark for the Windows version.



# POST CLICK

POST CLICK ( mouseX ; mouseY {; process} {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| mouseX | Longint | ⇒ | Horizontal coordinate |
| mouseY | Longint | ⇒ | Vertical coordinate |
| process | Longint | ⇒ | Destination process reference number, or Application event queue, if omitted, or 0 |
| * | | ⇒ | If specified, global coordinate system is used If omitted, local coordinate system is used |

## Description

The **POST CLICK** command simulates a mouse click. Its effect as if the user actually clicked the mouse button.

You pass the horizontal and vertical coordinates of the click in *mouseX* and *mouseY*. If you pass the * parameter, you express these coordinates relative to the screen. If you omit the * parameter, you express these coordinates relative to the frontmost window of the process whose process number you pass in *process*.

If you specify the *process* parameter, the click is sent to the process whose process number you pass in *process*. If you pass *0* (zero) or if you omit the parameter, the click is sent at the application level, and the 4D scheduler will dispatch it to the appropriate process.

# POST EVENT

POST EVENT ( what ; message ; when ; mouseX ; mouseY ; modifiers {; process} )

| Parameter | Type | | Description |
|---|---|---|---|
| what | Longint | ⇒ | Type of event |
| message | Longint | ⇒ | Event message |
| when | Longint | ⇒ | Event time expressed in ticks |
| mouseX | Longint | ⇒ | Horizontal coordinate of mouse |
| mouseY | Longint | ⇒ | Vertical coordinate of mouse |
| modifiers | Longint | ⇒ | Modifier keys state |
| process | Longint | ⇒ | Destination process reference number, or Application event queue, if omitted, or 0 |

## Description

The **POST EVENT** command simulates a keyboard or mouse event. Its effect is as if the user actually acted on the keyboard or the mouse.

You pass one of the following values in *what*:

| Constant | Type | Value |
|---|---|---|
| Auto key event | Longint | 5 |
| Key down event | Longint | 3 |
| Key up event | Longint | 4 |
| Mouse down event | Longint | 1 |
| Mouse up event | Longint | 2 |

If the event is a mouse-related event, you pass *0* (zero) in *message*. If the event is a keyboard-related event, you pass the code of the simulated character in *message*.

Usually, you pass the value returned by **Tickcount** in *when*.

If the event is a mouse-related event, you pass the horizontal and vertical coordinates of the click in *mouseX* and *mouseY*.

In the parameter *modifiers*, you pass one or a combination of the constants of the **Events (Modifiers)** theme.

| Constant | Type | Value | Comment |
|---|---|---|---|
| Activate window bit | Longint | 0 | |
| Activate window mask | Longint | 1 | |
| Caps lock key bit | Longint | 10 | Windows and OS X |
| Caps lock key mask | Longint | 1024 | Windows and OS X |
| Command key bit | Longint | 8 | Ctrl key under Windows, Command key under OS X |
| Command key mask | Longint | 256 | Ctrl key under Windows, Command key under OS X |
| Control key bit | Longint | 12 | Ctrl key under OS X, or right click under Windows and OS X |
| Control key mask | Longint | 4096 | Ctrl key under OS X, or right click under Windows and OS X |
| Mouse button bit | Longint | 7 | |
| Mouse button mask | Longint | 128 | |
| Option key bit | Longint | 11 | Alt key (also called Option under OS X) |
| Option key mask | Longint | 2048 | Alt key (also called Option under OS X) |
| Right control key bit | Longint | 15 | |
| Right control key mask | Longint | 32768 | |
| Right option key bit | Longint | 14 | |
| Right option key mask | Longint | 16384 | |
| Right shift key bit | Longint | 13 | |
| Right shift key mask | Longint | 8192 | |
| Shift key bit | Longint | 9 | Windows and OS X |
| Shift key mask | Longint | 512 | Windows and OS X |

For example, to simulate the Shift key, pass Shift key bit.

If you specify the *process* parameter, the event is sent to the process whose process number you pass in *process*. If you pass *0* (zero) or if you omit the parameter, the event is sent at the application level, and the 4D scheduler will dispatch it to the appropriate process.

# POST KEY

POST KEY ( code {; modifiers {; process}} )

| Parameter | Type | | Description |
|---|---|---|---|
| code | Longint | ⇒ | Character code or function key code |
| modifiers | Longint | ⇒ | State of modifier keys |
| process | Longint | ⇒ | Destination process reference number, or Application event queue, if omitted, or 0 |

## Description

The **POST KEY** command simulates a keystroke. Its effect is as if the user actually entered a character on the keyboard.

You pass the code of the character in *code*.

If you pass the *modifiers* parameter, you pass one or a combination of the **Events (Modifiers)** constants:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Activate window bit | Longint | 0 | |
| Activate window mask | Longint | 1 | |
| Caps lock key bit | Longint | 10 | Windows and OS X |
| Caps lock key mask | Longint | 1024 | Windows and OS X |
| Command key bit | Longint | 8 | Ctrl key under Windows, Command key under OS X |
| Command key mask | Longint | 256 | Ctrl key under Windows, Command key under OS X |
| Control key bit | Longint | 12 | Ctrl key under OS X, or right click under Windows and OS X |
| Control key mask | Longint | 4096 | Ctrl key under OS X, or right click under Windows and OS X |
| Mouse button bit | Longint | 7 | |
| Mouse button mask | Longint | 128 | |
| Option key bit | Longint | 11 | Alt key (also called Option under OS X) |
| Option key mask | Longint | 2048 | Alt key (also called Option under OS X) |
| Right control key bit | Longint | 15 | |
| Right control key mask | Longint | 32768 | |
| Right option key bit | Longint | 14 | |
| Right option key mask | Longint | 16384 | |
| Right shift key bit | Longint | 13 | |
| Right shift key mask | Longint | 8192 | |
| Shift key bit | Longint | 9 | Windows and OS X |
| Shift key mask | Longint | 512 | Windows and OS X |

For example, to simulate the Shift key, pass Shift key mask. If you do not pass *modifiers*, no modifiers are simulated.

If you specify the *process* parameter, the keystroke is sent to the process whose process number you pass in *process*. If you pass *0* (zero) or if you omit the parameter, the keystroke is sent at the application level, and the 4D scheduler will dispatch it to the appropriate process.

## Example

See example for the **Process number** command.

# REDRAW

REDRAW ( object )

| Parameter | Type | | Description |
|---|---|---|---|
| object | Form object | ⇒ | Table for which to redraw the subform, or Field for which to redraw the area, or Variable for which to redraw the area, or List box to be updated |

## Description

When you use a method to change the value of a field displayed in a subform, you must execute **REDRAW** to ensure that the form is updated.

In the context of list boxes in selection mode, when the **REDRAW** statement is applied to a list box type object it refreshes the data that is displayed in the object. This statement must be called typically after data modification has occurred in the records of the selection.

# SET ABOUT

SET ABOUT ( itemText ; method )

| Parameter | Type | | Description |
|---|---|---|---|
| itemText | String | ⇒ | New About menu item text |
| method | String | ⇒ | Name of method to execute when menu item is chosen |

## Description

The **SET ABOUT** command changes the **About 4D** menu command in the Help (Windows) or in the **Application** (Mac OS X) menu to *itemText*.

After the call, when a user selects this menu command in Design or Application mode, *method* will be called. Typically, this method can open a dialog box to give version information about your database.

This command is used with 4D (all modes), 4D Desktop and 4D Server. A new process is created when it is run on a server machine.

## Example 1

The following example replaces the **About** menu command with the **About Scheduler** menu command. The ***ABOUT*** method displays a custom About box:

```
SET ABOUT("About Scheduler…";"ABOUT")
```

## Example 2

The following example resets the About 4D menu command:

```
SET ABOUT("About 4D";"")
```

# SET CURSOR

SET CURSOR {( cursor )}

| Parameter | Type | | Description |
|---|---|---|---|
| cursor | Longint | ⇒ | Cursor resource number |

## Description

The **SET CURSOR** command changes the mouse cursor to the system cursor whose ID number you pass in *cursor*.

This command must be called in the context of the On Mouse Move **Form event**.

To restore the standard mouse cursor, call the command without any parameter.

Here are the cursors that can be passed in the *cursor* parameter:

| Number | Cursor |
|---|---|
| 1 |  |
| 2 |  |
| 4 |  |
| 9000 |  |
| 9001 |  |
| 9003 |  |
| 9004 |  |
| 9005 |  |
| 9006 |  |
| 9021 |  |
| 351 |  |
| 9010 |  |
| 9011 |  |
| 9013 |  |
| 9014 |  |
| 9015 |  |
| 9016 |  |
| 9017 |  |
| 9019 |  |
| 9020 |  |
| 559 |  |
| 560 |  |

**Note:** Cursor availability and appearance may vary depending on the operating system.

## Example

You want the cursor to be displayed as a  when the mouse moves over a variable area in the form. In the object method of the variable, you can write:

```
If(Form event=On Mouse Move)
   SET CURSOR(9019)
End if
```

# SET FIELD TITLES

SET FIELD TITLES ( aTable ; fieldTitles ; fieldNumbers {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ⇒ | Table for which to set the field titles |
| fieldTitles | String array | ⇒ | Field names as they must appear in dialog boxes |
| fieldNumbers | Longint array | ⇒ | Actual field numbers |
| * | | ⇒ | Use the custom names in the formula editor |

## Description

**SET FIELD TITLES** lets you mask, rename, and reorder the fields of the table passed in *aTable* or *aSubtable* when they appear in standard 4D dialog boxes, such as the Query editor, within the Application environment (more specifically, when the editors are called using 4D language commands).

Using this command, you can also rename on the fly the labels of the fields in your forms, if you used dynamic names. For more information about inserting dynamic field and table names in forms, refer to the 4D Design Reference manual.

The *fieldTitles* and *fieldNumbers* arrays must be synchronized. In the *fieldTitles* array, you pass the name of the fields as you would like them to appear. If you do not want to show a particular field, do not include its name or new title in the array. The fields appear in the order you specify in this array. In each element of the *fieldNumbers* array, you pass the actual field number corresponding to the field name or new title passed in the same element number in the *fieldTitles* array.

For example, you have a table or subtable composed of the fields F, G, and H, created in that order. You want these fields to appear as M, N, and O. In addition you do not want to show field N. Finally, you want to show O and M in that order. To do so, pass O and M in a two-element *fieldTitles* array and pass 3 and 1 in a two-element *fieldNumbers* array.

The optional * parameter indicates whether or not custom names defined using this command can be used in 4D formulas.

- By default, when this parameter is omitted, formulas executed in 4D cannot use these custom names; it is necessary to use the real table names. Using custom names gives you greater freedom for naming fields since the language interpreter does not process custom names
- If the * parameter is passed, the names defined by this command are used in the formulas executed by 4D. **Be careful in this case**, the custom names must not contain characters that are “forbidden” by the 4D language interpreter, like -? *! (for more information, refer to the “**Identifiers**” section).

**Note:** If your application provides access to the formula editor (for example via the Quick report editor), you must pass the * parameter in order to ensure interface consistency.

**SET FIELD TITLES** does NOT change the actual structure of your table. It only affects subsequent uses of the standard 4D dialog boxes and forms using dynamic names when they are called using language commands (the real structure of the database is displayed when the editor or form is called from a menu command in Design mode). The scope of the **SET FIELD TITLES** command is the worksession. One benefit in Client/Server mode is that several remote 4D stations can simultaneously “see” your table in different ways. You can call **SET FIELD TITLES** as many times as you want.

Use the **SET FIELD TITLES** command for:

- Dynamically localizing a table.
- Showing fields the way you want, independent of the actual definition of your table.
- Showing fields in a way that depends on the identity or custom privileges of a user.

**WARNING:**

- **SET FIELD TITLES** does NOT override the Invisible property of a field. When a field is set to be invisible at the Design level of your database, even though it is included in a call to **SET FIELD TITLES**, it will not appear in Application mode.
- Plug-ins always access the "virtual" structure as specified by this command.

## Example

See example for the **SET TABLE TITLES** command.

# SET TABLE TITLES

SET TABLE TITLES ( tableTitles ; tableNumbers {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| tableTitles | String array | ⇒ | Table names as they must appear in dialog boxes |
| tableNumbers | Longint array | ⇒ | Actual table numbers |
| * | | ⇒ | Use the custom names in the formula editor |

## Description

**SET TABLE TITLES** lets you mask, rename, and reorder the tables of your database that appear in standard 4D dialog boxes within the Application environment (when these editors are called using 4D language commands). For example, this command can modify the display of tables in the Query editor in Application mode.
Using this command, you can also rename on the fly the table labels in your forms, if you used dynamic names. For more information about inserting dynamic field and table names in the forms, refer to **Using references in static text** in the *4D Design Reference* manual.

The *tableTitles* and *tableNumbers* arrays must be synchronized. In the *tableTitles* array, you pass the names of the tables as you would like them to appear. If you do not want to show a particular table, do not include its name or new title in the array. The tables appear in the order you specify in this array. In each element of the *tableNumbers* array, you pass the actual table number corresponding to the table name or new title passed in the same element number in the *tableTitles* array.

For example, you have a database composed of the tables A, B, and C, created in that order. You want these tables to appear as X, Y, and Z. In addition you do not want to show table B. Finally, you want to show Z and X, in that order. To do so, you pass Z and X in a two-element *tableTitles* array, and you pass 3 and 1 in a two-element *tableNumbers* array.

The optional * parameter indicates whether or not custom names defined using this command are used in 4D formulas:

- By default, when this parameter is omitted, formulas executed in 4D cannot use these custom names; it is necessary to use the real table names. Using custom names gives you greater freedom for naming tables since the language interpreter does not process custom names.
- If the * parameter is passed, the names defined by this command are used in the formulas executed by 4D. **Be careful in this case**, the custom names must not contain characters that are “forbidden” by the 4D language interpreter, like -? *%! For example, the name "Rate_in_%" could not be used in a formula (for more information, refer to the **Identifiers** section).

**Note:** If your application provides access to the formula editor (for example via the Quick report editor), it is necessary to pass the * parameter in order to ensure interface consistency.

**SET TABLE TITLES** does NOT change the actual structure of your database. It only affects subsequent uses of the standard 4D dialog boxes and forms using dynamic names when they are called via language commands (the real structure of the database is displayed when the editor or form is called from a menu command in Design mode). The scope of the **SET TABLE TITLES** command is the worksession. One benefit in Client/Server is that several 4D Client stations can simultaneously “see” your database in different ways. You can call **SET TABLE TITLES** as many times as you want.

Use the **SET TABLE TITLES** command for:

- Dynamically localizing a database.
- Showing tables the way you want, independent from the actual definition of your database.
- Showing tables in a way that depends on the identity or custom privileges of a user.

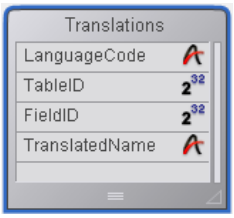
**Notes:**

- **SET TABLE TITLES** does NOT override the Invisible property of a table. When a table is set to be invisible at the structure level of your database, even though it is included in a call to **SET TABLE TITLES**, it will not appear in Application mode.
- Plug-ins always access the "virtual" structure as specified by this command.

## Example

You are building a 4D application that you plan to sell internationally. Therefore, you must carefully consider localization issues. Regarding the standard 4D dialog boxes that can appear in the Application environment and your forms that use dynamic names, you can address localization needs by using a [Translations] table and a few project methods to create and use fields localized for any number of countries.

In your database, add the following table:




Then, create the ***TRANSLATE TABLES AND FIELDS*** project method listed below. This method browses the actual structure of your database and creates all the necessary *[Translations]* records for the localization corresponding to the language passed as parameter.

```
 //TRANSLATE TABLES AND FIELDS project method
 //TRANSLATE TABLES AND FIELDS (Text)
 //TRANSLATE TABLES AND FIELDS (LanguageCode)

C_TEXT($1) //language code
C_LONGINT($vlTable;$vlField)
C_TEXT($Language)
$Language:=$1

For($vlTable;1;Get last table number) //Pass through each table
   If($vlTable#(Table(->[Translations]))) //Do not translate table of translations
 //Check if there is a translation of the table name for the specified language
      QUERY([Translations];[Translations]LanguageCode=$Language;*) //desired language
      QUERY([Translations]; & ;[Translations]TableID=$vlTable;*) //table number
      QUERY([Translations]; & ;[Translations]FieldID=0) //field number = 0 means that it is a table name
      If(Is table number valid($vlTable)) //check that the table still exists
         If(Records in selection([Translations])=0)
 //Otherwise, create the record
            CREATE RECORD([Translations])
            [Translations]LanguageCode:=$Language
            [Translations]TableID:=$vlTable
            [Translations]FieldID:=0
 //The name of the translated table will need to be entered
            [Translations]Translation:=Table name($vlTable)+" in "+$Language
            SAVE RECORD([Translations])
         End if

         For($vlField;1;Get last field number($vlTable))
 //Check if there is a translation of the field name for the specified language
            QUERY([Translations];[Translations]LanguageCode=$Language;*) //desired language
            QUERY([Translations]; & ;[Translations]TableID=$vlTable;*) //table number
            QUERY([Translations]; & ;[Translations]FieldID=$vlField) //field number
            If(Is field number valid($vlTable;$vlField))
               If(Records in selection([Translations])=0)
 //Otherwise, create the record
                  CREATE RECORD([Translations])
                  [Translations]LanguageCode:=$Language
                  [Translations]TableID:=$vlTable
                  [Translations]FieldID:=$vlField
 //The name of the translated field will need to be entered
                  [Translations]Translation:=Field name($vlTable;$vlField)+" in "+$Language
                  SAVE RECORD([Translations])
               End if
            Else
               If(Records in selection([Translations])#0)
 //If the field no longer exists, remove the translation
                  DELETE RECORD([Translations])
               End if
            End if
         End for
      Else
```

```
            If(Records in selection([Translations])#0)
   //If the table no longer exists, remove the translation
               DELETE RECORD([Translations])
            End if
         End if
      End if
   End for
```

At this point, if you execute the following line, you create as many records as needed for a Spanish localization of the tables and fields titles.

```
TRANSLATE TABLES AND FIELDS("Spanish")
```

After this call has been executed, you can then enter the *[Translations]Translated Name* for each of the newly created records.

Finally, each time you want to show your database's standard 4D dialog boxes or forms with dynamic titles using the Spanish localization, you execute the following line:

```
LOCALIZED TABLES AND FIELDS("Spanish")
```

with the project method ***LOCALIZED TABLES AND FIELDS***:

```
   //LOCALIZED TABLES AND FIELDS global method
   //LOCALIZED TABLES AND FIELDS (Text)
   //LOCALIZED TABLES AND FIELDS (LanguageCode)

C_TEXT($1) //Language code
C_LONGINT($vlTable;$vlField)
C_TEXT($Language)
C_LONGINT($vlTableNum;$vlFieldNum)
$Language:=$1

   //Updating table names
ARRAY TEXT($asNames;0) //Initialize arrays for SET TABLE TITLES and SET FIELD TITLES
ARRAY INTEGER($aiNumbers;0)
QUERY([Translations];[Translations]LanguageCode=$Language;*)
QUERY([Translations]; & ;[Translations]FieldID=0) //thus table names
SELECTION TO ARRAY([Translations]Translation;$asNames;[Translations]TableID;$aiNumbers)
SET TABLE TITLES($asNames;$aiNumbers)

   //Updating field names
$vlTableNum:=Get last table number //Get number of tables in database
For($vlTable;1;$vlTableNum) //Pass through each table
   If(Is table number valid($vlTable))
      QUERY([Translations];[Translations]LanguageCode=$Language;*)
      QUERY([Translations]; & ;[Translations]TableID=$vlTable;*)
      QUERY([Translations]; & ;[Translations]FieldID#0) //avoid the zero that serves as table name
      SELECTION TO ARRAY([Translations]Translation;$asNames;[Translations]FieldID;$aiNumbers)
      SET FIELD TITLES(Table($vlTable)->;$asNames;$aiNumbers)
   End if
End for
```

Note that new localizations can be added to the database without modifying or recompiling the code.

# Shift down

Shift down -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | State of the Shift key |

## Description

**Shift down** returns TRUE if the Shift key is pressed.

## Example

The following object method for the button *bAnyButton* performs different actions, depending on which modifier keys are pressed when the button is clicked:

```
  ` bAnyButton Object Method
Case of
  ` Other multiple key combinations could be tested here
  ` ...
  :(Shift down&Windows Ctrl down)
  ` Shift and Windows Ctrl (or Macintosh Command) keys are pressed
     DO ACTION1
  ` ...
  :(Shift down)
  ` Only Shift key is pressed
     DO ACTION2
  ` ...
  :(Windows Ctrl down)
  ` Only Windows Ctrl (or Macintosh Command) key is pressed
     DO ACTION3
  ` ...
  ` Other individual keys could be tested here
  ` ...
End case
```

# SHOW MENU BAR

SHOW MENU BAR
Does not require any parameters

## Description

The **SHOW MENU BAR** command makes the menu bar visible.
If the menu bar was already visible, the command does nothing.

## Example

See example for the **HIDE MENU BAR** command.

# Windows Alt down

Windows Alt down -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Boolean | State of the Windows Alt key (Option key on Macintosh) |

## Description

**Windows Alt down** returns TRUE if the Windows Alt key is pressed.

**Note:** When called on a Macintosh platform, **Windows Alt down** returns TRUE if the Macintosh Option key is pressed.

## Example

See example for the **Shift down** command.

# Windows Ctrl down

Windows Ctrl down -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | Boolean | State of the Windows Ctrl key (Command key on Macintosh) |

## Description

**Windows Ctrl down** returns TRUE if the Windows Ctrl key is pressed.

**Note:** When called on a Macintosh platform, **Windows Ctrl down** returns TRUE if the Macintosh Command key is pressed.

## Example

See example for the **Shift down** command.

# _o_Get platform interface

_o_Get platform interface -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ⮌ | Current platform interface in use |

## Compatibility Note

The platform interface is now managed automatically by 4D. This command should not be used since it no longer returns significant values.

# _o_INVERT BACKGROUND

_o_INVERT BACKGROUND ( {* ;} textObject )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | Allows entry of a variable or object name |
| textObject | Variable, Field | ⇒ | Text variable or field to invert |

## Description

This command is no longer supported in 4D.

# _o_SET PLATFORM INTERFACE

_o_SET PLATFORM INTERFACE ( interface )

| Parameter | Type | | Description |
|---|---|---|---|
| interface | Longint | ⇒ | New platform interface setting: -1 Automatic 0 Mac OS 7 1 Windows 3.11, NT 3.51 2 Windows 9x 3 Mac OS 9 4 Mac Theme |

## Compatibility Note

The platform interface is managed automatically by 4D. This command is now ignored and must no longer be used.

# Users and Groups

- BLOB TO USERS
- CHANGE CURRENT USER
- CHANGE LICENSES
- CHANGE PASSWORD
- Current user
- DELETE USER
- EDIT ACCESS
- Get default user
- GET GROUP LIST
- GET GROUP PROPERTIES
- Get plugin access
- GET USER LIST
- GET USER PROPERTIES
- Is license available
- Is user deleted
- Set group properties
- SET PLUGIN ACCESS
- Set user properties
- User in group
- USERS TO BLOB
- Validate password

# BLOB TO USERS

BLOB TO USERS ( users )

| Parameter | Type | | Description |
|---|---|---|---|
| users | BLOB | ⇒ | BLOB (encrypted) containing database user accounts created and saved by the database Administrator |

## Description

The **BLOB TO USERS** command replaces the user accounts and groups created by the Administrator in the database with the ones found in the BLOB *users*. The BLOB *users* is encrypted and must have been created using the **USERS TO BLOB** command.

Only the database Administrator or Designer can execute this command. If another user attempts to execute it, the command does nothing and a privilege error (-9949) is generated.

This command causes the replacement of any existing accounts and groups created by the Administrator in the database. If the BLOB *users* contains valid data, the command performs the following operations:

- all users and groups defined in the database whose reference numbers are negative (groups and users created by the Administrator) are removed from the structure,
- all users and groups found in the BLOB *users* are added to the structure.

**Compatibility note:** User and group files (.4UG extension) created by the **Save Groups...** menu command in versions of 4D prior to 2004 can be loaded in 4D version 2004 and higher using the following sequence:

```
DOCUMENT TO BLOB(mydoc;blob)
BLOB TO USERS(blob)
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

# CHANGE CURRENT USER

CHANGE CURRENT USER {( user ; password )}

| Parameter | Type | | Description |
|---|---|---|---|
| user | String, Longint | ➡ | Name or unique user ID |
| password | String | ➡ | Password (not encrypted) |

## Description

**CHANGE CURRENT USER** changes the identity of the current user in the database, without needing to quit. The user can change their identity themselves either using the database connection dialog box (when the command is called without parameters) or directly via the command. When a user changes their identity, they abandon any former access privileges in favor of those belonging to the chosen user.

If the **CHANGE CURRENT USER** command is executed without parameters, the database connection dialog box appears. The user must then enter or select a valid name and password in order to enter the database. The contents of the connection dialog box will depend on the options set on the **Security** page of the Database Settings.

**Note:** The access control system must be enabled in order to use this command, i.e. a password must have been assigned to the *Designer*. Otherwise, **CHANGE CURRENT USER** has no effect and does not display the standard window for switching users.

You can also pass the two optional *user* and *password* parameters to specify by programming the new account to be used. In the *user* parameter, pass the name or unique user ID (*userRef*) of the account to be used. The user names and IDs can be obtained using the **GET USER LIST** command.

| User ID | User description |
|---|---|
| 1 | Designer |
| 2 | Administrator |
| 3 to 15000 | User created by the Designer<br>(user No. 3 is the first user created by the Designer, user No. 4 is the second, etc.). |
| -11 to -15010 | User created by the Administrator<br>(user No. -11 is the first user created by the Administrator, user No. -12 is the second, etc.). |

If the user account does not exist or was deleted, error -9979 is returned. You can intercept this error with the error-handling method installed by the **ON ERR CALL** command. Otherwise, you can call the function **Is user deleted** to test the user account before calling this command.

Pass the non-encrypted user account password in the *password* parameter. If the password does not match the user, the command will return error message -9978 and do nothing.

The command execution is now delayed to prevent flooding (brute force attack), in other words, attempts of multiple user name/password combinations. As a result, after the 4th call to this command, it is run only after a period of 10 seconds. This delay is throughout the entire work station.

### Offering a custom access management dialog box

The **CHANGE CURRENT USER** command can be used to set up custom dialog boxes for entering the name and password (with entry and expiration rules) that benefit from the same advantages as the access control system of 4D.
Here is how It works:

**1.** The database is entered directly in the “Default user” mode, without a dialog box.

**2.** The **On Startup database method** displays a custom dialog box for entering the user name and password. All types of processing are foreseeable in the dialog box:
- You can display the list of database users, as in the standard access dialog box of 4D, using the **GET USER LIST** command.
- The password entry field can contain various controls to check the validity of the entered characters (minimum number of characters, uniqueness, etc.).
- If you want the characters of passwords being entered to be masked on screen, you can use the **FILTER KEYSTROKE** command with the special *%password* font.
- Expiration rules can be applied at the moment when the dialog box is validated: expiration date, forced change to the initial connection, locking of account after several incorrect entries, memorization of passwords already used, etc.

**3.** When the entry is validated, the required information (user name and password) are passed to the **CHANGE CURRENT USER** command in order to open the database with the user account privileges.

## Example

The following example displays the connection dialog box:

```
CHANGE CURRENT USER
```

# CHANGE LICENSES

CHANGE LICENSES
Does not require any parameters

## Description

The **CHANGE LICENSES** command displays the 4D License Manager dialog box.

This command can only be used with 4D single-user applications and cannot be called from a component. When passwords are enabled, this command can only be executed by the Designer or Administrator; it does nothing when it is called by users that do not have appropriate access rights.

Using the License Manager dialog box, a user can activate plug-ins or the Web server on the machine where it is executed. In 4D, you can display this dialog box by selecting the **License Manager...** command in the **Help** menu.

**CHANGE LICENSES** is a convenient way to activate licenses and add expansion numbers in a compiled single-user 4D application distributed to customers. 4D developers or IS managers can use this command to distribute a 4D application and let users enter their license without sending an update of the application each time.

For more information about this dialog box, refer to the **Installation and activation** section in the *4D Installation Guide*.

## Example

In a custom configuration or preferences dialog box, you include a button whose associated method is:

```
 // Object method for bLicense button
CHANGE LICENSES
```

This way a user can activate licenses without having to modify the database.

# CHANGE PASSWORD

CHANGE PASSWORD ( password )

| Parameter | Type | | Description |
|---|---|---|---|
| password | String | ➡ | New password |

## Description

**CHANGE PASSWORD** changes the password of the current user. This command replaces the current password with the new password you pass in *password*.

**Warning:** Password are case-sensitive.

## Example

The following example allows the user to change his or her password.

```
CHANGE CURRENT USER ` Present user with password dialog
If(OK=1)
   $pw1:=Request("Enter new password for "+Current user)
 ` The password should be at least five characters long
   If(((OK=1)&($pw1#""))&(Length($pw1)>5))
 ` Make sure the password has been entered correctly
      $pw2:=Request("Enter password again")
      If((OK=1)&($pw1=$pw2))
         CHANGE PASSWORD($pw2) ` Change the password
      End if
   End if
End if
```

# Current user

Current user -> Function result

| Parameter | Type | Description |
|---|---|---|
| Function result | String | User name of the current user |

## Description

**Current user** returns the user name of the current user.

## Example

See example for the **User in group** command.

# DELETE USER

DELETE USER ( userID )

| Parameter | Type | | Description |
|---|---|---|---|
| userID | Longint | ➡ | ID number of user to delete |

## Description

The **DELETE USER** command deletes the user whose unique user ID number you pass in *userID*. You must pass a valid user ID number returned by the **GET USER LIST** command.

If the user account does not exist or has already been deleted, the error -9979 is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

Only the Designer and Administrator can delete users. It is not possible for the Administrator to delete a user created by the Designer.

Deleted user names no longer appear in the Users editor displayed when you call **EDIT ACCESS**, nor in the Design mode. Note that the numbers for deleted users can be reassigned when new user accounts are created.

## Error management

If you do not have the proper access privileges for calling **DELETE USER** or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# EDIT ACCESS

EDIT ACCESS
Does not require any parameters

## Description

**EDIT ACCESS** lets the user edit the password system. When this command is executed, the Toolbox window with only the Users and User groups pages appears.

**Note:** This command opens a modal window so you must not call it from another modal window; if you do, it will not do anything.

Groups can be edited by the Designer, the Administrator and group owners. The Designer and the Administrator can edit any group. Group owners can edit only the groups they own. Users can be added to and removed from groups. The command has no effect if no groups are defined.

The Designer and the Administrator can add new users, as well as assign them to groups.

## Example

The following example displays the Users and User groups management window to the user:

```
EDIT ACCESS
```

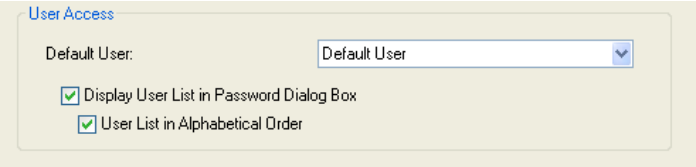
# Get default user

Get default user -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ⮌ | Unique user ID number |

## Description

The **Get default user** command returns the unique user ID of the user set as “Default user” in the database Preferences dialog box:




The following numbers can be used as user IDs:

| ID | User description |
|---|---|
| 1 | Designer |
| 2 | Administrator |
| 3 to 15000 | User created by Designer (user #3 is the 1st user created by Designer, user #4 is the second, and so on). |
| -11 to -15010 | User created by the Administrator (user #-11 is the 1st user created by Administrator, user #-12 is the second, and so on). |

If no default user has been set, the command returns 0.

# GET GROUP LIST

GET GROUP LIST ( groupNames ; groupNumbers )

| Parameter | Type | | Description |
|---|---|---|---|
| groupNames | String array | ⇐ | Names of the groups as they appear in the Password editor window |
| groupNumbers | Longint array | ⇐ | Corresponding unique group ID numbers |

## Description

**GET GROUP LIST** populates the arrays *groupNames* and *groupNumbers* with the names and unique ID numbers of the groups as they appear in the Password editor window.

The array *groupNumbers*, synchronized with *groupNames*, is filled with the corresponding unique group ID numbers. These numbers can have the following ranges:

| Group ID number | Group description |
|---|---|
| 15001 to 32767 | Group created by the Designer or affiliated Group Owner<br>(group #15001 is the first group created by the Designer,group #15002 the second, and so on). |
| -15001 to -32768 | Group created by the Administrator or affiliated Group Owner<br>(group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on). |

## Error management

If you do not have the proper access privileges for calling **GET GROUP LIST** or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# GET GROUP PROPERTIES

GET GROUP PROPERTIES ( groupID ; name ; owner {; members} )

| Parameter | Type | | Description |
|---|---|---|---|
| groupID | Longint | ⇒ | Unique group ID number |
| name | String | ⇐ | Name of the group |
| owner | Longint | ⇐ | User ID number of group owner |
| members | Longint array | ⇐ | Group members |

## Description

**GET GROUP PROPERTIES** returns the properties of the group whose unique group ID number you pass in *groupID*. You must pass a valid group ID number returned by the command **GET GROUP LIST**. Group ID numbers can have the following values or ranges:

| Group ID number | Group description |
|---|---|
| 15001 to 32767 | Group created by the Designer or affiliated Group Owner<br>(group #15001 is the first group created by the Designer, group #15002 the second, and so on). |
| -15001 to -32768 | Group created by the Administrator or affiliated Group Owner<br>(group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on). |

If you do not pass a valid group ID number, **GET GROUP PROPERTIES** returns empty parameters.

After the call, you retrieve the name and owner of the group, in the parameters *name* and *owner*.

If you pass the optional *members* parameter, the unique ID numbers of the users and groups belonging to the group are returned. Member ID numbers can have the following ranges:

| Member ID number | Member Description |
|---|---|
| 1 | Designer user |
| 2 | Administrator user |
| 3 to 15000 | User created by the Designer of the database<br>(user #3 is the first user created by the Designer, user #4 the second, and so on). |
| -11 to -15000 | User created by the Administrator of the database<br>(user #-11 is the first user created by the Designer, user #-12 is the second, and so on). |
| 15001 to 32767 | Group created by the Designer or affiliated Group Owner<br>(group #15001 is the first group created by the Designer, group #15002 the second, and so on). |
| -15001 to -32768 | Group created by the Administrator or affiliated Group Owner<br>(group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on). |

## Error management

If you do not have the proper access privileges for calling **GET GROUP PROPERTIES** or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# Get plugin access

Get plugin access ( plugIn ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| plugIn | Longint | ⇒ | Plug-in number |
| Function result | String | ⇐ | Group name associated with plug-in |

## Description

The **Get plugin access** command returns the name of the user group authorized to use the plug-in whose number was passed in the *plugIn* parameter. If there is no group associated with the plug-in, the command returns an empty string ("").

Pass the number of the plug-in for which you want to find out the associated group of users in the *plugIn* parameter. Plug-in licenses include 4D Client Web and SOAP licenses. You can pass one of the following constants found in the **Is License Available** theme:

| Constant | Type | Value |
|---|---|---|
| 4D Client SOAP license | Longint | 808465465 |
| 4D Client Web license | Longint | 808465209 |
| 4D Draw license | Longint | 808464694 |
| 4D for ADO license | Longint | 808465714 |
| 4D for MySQL license | Longint | 808465712 |
| 4D for OCI license | Longint | 808465208 |
| 4D for PostgreSQL license | Longint | 808465713 |
| 4D for Sybase license | Longint | 808465715 |
| 4D ODBC Pro license | Longint | 808464946 |
| 4D View license | Longint | 808465207 |
| 4D Write license | Longint | 808464697 |

# GET USER LIST

GET USER LIST ( userNames ; userNumbers )

| Parameter | Type | | Description |
|---|---|---|---|
| userNames | String array | ⇦ | User names as they appear in the Password editor window |
| userNumbers | Longint array | ⇦ | Corresponding unique user ID numbers |

## Description

**GET USER LIST** populates the arrays *userNames* and *userNumbers* with the names and unique ID numbers of the users as they appear in the Passwords window.

The array *userNames* is filled with the user names displayed in the Passwords window, including users whose accounts are disabled (user names displayed in green in the Passwords window).

**Note:** Use the **Is user deleted** command to detect deleted users.

The array *userNumbers*, synchronized with *userNames*, is filled with the corresponding unique user ID numbers. These numbers can have the following values or ranges:

| User ID number | User description |
|---|---|
| 1 | Designer user |
| 2 | Administrator user |
| 3 to 15000 | User created by the Designer of the database<br>(user #3 is the first user created by the Designer, user #4 the second, and so on). |
| -11 to -15000 | User created by the Administrator of the database<br>(user #-11 is the first user created by the Designer, user #-12 is the second, and so on). |

## Error management

If you do not have the proper access privileges for calling **GET USER LIST** or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# GET USER PROPERTIES

GET USER PROPERTIES ( userID ; name ; startup ; password ; nbLogin ; lastLogin {; memberships {; groupOwner}} )

| Parameter | Type | | Description |
|---|---|---|---|
| userID | Longint | ⇒ | Unique user ID number |
| name | String | ⇐ | Name of the user |
| startup | String | ⇐ | Startup method name |
| password | String | ⇐ | Always an empty string |
| nbLogin | Longint | ⇐ | Number of logins to the database |
| lastLogin | Date | ⇐ | Date of last login to the database |
| memberships | Longint array | ⇐ | ID numbers of groups to which the user belongs |
| groupOwner | Longint | ⇐ | ID number of user group owner |

## Description

**GET USER PROPERTIES** returns the information about the user whose unique user ID number you pass in *userID*. You must pass a valid user ID number returned by the **GET USER LIST** command.

If the user account does not exist or has been deleted, the error -9979 is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**. Otherwise, you can call **Is user deleted** to test the user account before calling **GET USER PROPERTIES**.

User ID numbers can have the following values or ranges:

| User ID number | User description |
|---|---|
| 1 | Designer user |
| 2 | Administrator user |
| 3 to 15000 | User created by the Designer of the database<br>(user #3 is the first user created by the Designer, user #4 the second, and so on). |
| -11 to -15000 | User created by the Administrator of the database<br>(user #-11 is the first user created by the Designer, user #-12 is the second, and so on). |

After the call, you retrieve the name, startup method, encrypted password, number of logins and date of last login for the user, in the parameters *name*, *startup*, *password*, *nbLogin* and *lastLogin*.

**Note:** The *password* parameter is obsolete (it always returns an empty string). If you want to check a user's password, use the **Validate password** function.

If you pass the optional *memberships* parameter, the unique ID numbers of the groups to which the user belongs are returned. Group ID numbers can have the following ranges:

If you pass the optional *groupOwner* parameter, you get the ID number of the user group “owner”, i.e. the default owner group of the objects created by this user.

The group ID numbers can be the following:

| Group ID number | Group description |
|---|---|
| 15001 to 32767 | Group created by the Designer or affiliated Group Owner<br>(group #15001 is the first group created by the Designer, group #15002 the second, and so on). |
| -15001 to -32768 | Group created by the Administrator or affiliated Group Owner<br>(group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on). |

## Error management

If you do not have the proper access privileges for calling **GET USER PROPERTIES** or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# Is license available

Is license available {( license )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| license | Longint | ⇒ | Plug-in for which license validity testing is desired |
| Function result | Boolean | ⇐ | True if plug-in is available, otherwise False |

## Description

The **Is license available** command lets you find out the availability of a plug-in. It is useful, for instance, for displaying or hiding functions requiring the presence of a plug-in.

The **Is license available** command can be used in three different ways:

- The *license* parameter is omitted: in this case, the command returns **False** if the 4D application is in demonstration mode.
- You pass one of the constants of the "**Is License Available**" theme in the *license* parameter:

| Constant | Type | Value |
|---|---|---|
| 4D Client SOAP license | Longint | 808465465 |
| 4D Client Web license | Longint | 808465209 |
| 4D Draw license | Longint | 808464694 |
| 4D for ADO license | Longint | 808465714 |
| 4D for MySQL license | Longint | 808465712 |
| 4D for OCI license | Longint | 808465208 |
| 4D for PostgreSQL license | Longint | 808465713 |
| 4D for Sybase license | Longint | 808465715 |
| 4D Mobile license | Longint | 808464439 |
| 4D Mobile Test license | Longint | 808465719 |
| 4D ODBC Pro license | Longint | 808464946 |
| 4D SOAP license | Longint | 808465464 |
| 4D View license | Longint | 808465207 |
| 4D Web license | Longint | 808464945 |
| 4D Write license | Longint | 808464697 |

  In this case, the command returns **True** if the corresponding plug-in has a license available. The command takes into account any licenses attributed in Design mode or via the **SET PLUGIN ACCESS** command.
  **Is license available** returns **False** if the plug-in is operating in demo mode.
- You pass the ID number of the plug-in "4BNX" resource directly in the license parameter. In this case, the command behaves as described above.

# Is user deleted

Is user deleted ( userNumber ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| userNumber | Longint | ⇒ | User ID number |
| Function result | Boolean | ⇐ | TRUE = User account is deleted or does not exist FALSE = User account is active |

## Description

The **Is user deleted** command tests the user account whose unique user ID number you pass in *userID*.

If the user account does not exist or has been deleted, **Is user deleted** returns TRUE. Otherwise, it returns FALSE.

## Error management

If you do not have the proper access privileges for calling **Is user deleted** or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# Set group properties

Set group properties ( groupID ; name ; owner {; members} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| groupID | Longint | ⇒ | Unique ID number of group, or -1 for adding a Designer group, or -2 for adding an Administrator group |
| name | String | ⇒ | New group name |
| owner | Longint | ⇒ | User ID number of new group owner |
| members | Longint array | ⇒ | New group members |
| Function result | Longint | ⇐ | Unique ID number of new group |

## Description

**Set group properties** enables you to change and update the properties of an existing group whose unique group ID number you pass in *groupID*, or to add a new group affiliated with the Designer or the Administrator.

If you are changing the properties of an existing group, you must pass a valid group ID number returned by the command **GET GROUP LIST**. Group ID numbers can have the following values or ranges:

| Group ID number | Group description |
|---|---|
| 15001 to 32767 | Group created by the Designer or affiliated Group Owner<br>(group #15001 is the first group created by the Designer, group #15002 the second, and so on). |
| -15001 to -32768 | Group created by the Administrator or affiliated Group Owner<br>(group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on). |

To add a new group affiliated with the Designer, pass -1 in *groupID*. To add a new group affiliated with the Administrator, pass -2 in *groupID*. After the call, if the group is successfully added, its unique ID number is returned in *groupID*.

If you do not pass -1, -2 or a valid group ID number, **Set group properties** does nothing.

Before the call, you pass the new name and owner of the group in the parameters *name* and *owner*.If you do not want to change all the properties of the group (besides the members, see below), first call **GET GROUP PROPERTIES** and pass the returned values for the properties you want to leave unchanged.

If you do not pass the optional *members* parameter, the current member list of the group is left unchanged. If you do not pass *members* while adding a group, the group will have no members.

**Note:** The group owner is not automatically set as a member of the group that he or she owns. It is up to you to include the group owner in the group, using the *members* parameter.

If you pass the optional *members* parameter, you change the whole member list for the group. Before the call, you must populate the array *members* with the unique ID numbers of the users and groups the group will get as members. Member ID numbers can have the following ranges:

| Member ID number | Member Description |
|---|---|
| 1 | Designer user |
| 2 | Administrator user |
| 3 to 15000 | User created by the Designer of the database<br>(user #3 is the first user created by the Designer, user #4 the second, and so on). |
| -11 to -15000 | User created by the Administrator of the database<br>(user #-11 is the first user created by the Designer, user #-12 is the second, and so on). |
| 15001 to 32767 | Group created by the Designer or affiliated Group Owner<br>(group #15001 is the first group created by the Designer, group #15002 the second, and so on). |
| -15001 to -32768 | Group created by the Administrator or affiliated Group Owner<br>(group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on). |

To remove all the members from a group, pass an empty *members* array.

## Error management

If you do not have the proper access privileges for calling **Set group properties** or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# SET PLUGIN ACCESS

SET PLUGIN ACCESS ( plugIn ; group )

| Parameter | Type | | Description |
|---|---|---|---|
| plugIn | Longint | ⇒ | Plug-in number |
| group | String | ⇒ | Group name to associate with plug-in |

## Description

The **SET PLUGIN ACCESS** command sets, by programming, the user group allowed to use each “serialized” plug-in that is installed in the database. This way you can manage how plug-in licenses are used.

**Note:** This can also be done in the Design environment using the Groups editor.

Pass the number of the plug-in to be associated with a group of users in the *plugIn* parameter. Plug-in licenses include 4D Client Web and SOAP licenses. You can pass one of the following constants found in the **Is License Available** theme:

| Constant | Type | Value |
|---|---|---|
| 4D Client SOAP license | Longint | 808465465 |
| 4D Client Web license | Longint | 808465209 |
| 4D Draw license | Longint | 808464694 |
| 4D for ADO license | Longint | 808465714 |
| 4D for MySQL license | Longint | 808465712 |
| 4D for OCI license | Longint | 808465208 |
| 4D for PostgreSQL license | Longint | 808465713 |
| 4D for Sybase license | Longint | 808465715 |
| 4D ODBC Pro license | Longint | 808464946 |
| 4D View license | Longint | 808465207 |
| 4D Write license | Longint | 808464697 |

Pass the name of the group whose users are authorized to use the plug-in in *group*.

**Note:** Only one group at a time can be allowed to use a plug-in. When this command is executed, if another group had the plug-in access rights, it loses this privilege.

# Set user properties

Set user properties ( userID ; name ; startup ; password ; nbLogin ; lastLogin {; memberships {; groupOwner}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| userID | Longint | ➡ | Unique ID number of user account, or -1 for adding a user affiliated with the Designer, or -2 for adding a user affiliated with the Administrator |
| name | String | ➡ | New user name |
| startup | String | ➡ | Name of new user startup method |
| password | String | ➡ | New (unencrypted) password, or * to leave the password unchanged |
| nbLogin | Longint | ➡ | New number of logins to the database |
| lastLogin | Date | ➡ | New date of last login to the database |
| memberships | Longint array | ➡ | ID numbers of groups to which the user belongs |
| groupOwner | Longint | ➡ | Reference number of user group owner |
| Function result | Longint | ⬅ | Unique ID number of new user |

## Description

**Set user properties** lets you change and update the properties of an existing user account whose unique user ID number you pass in *userID*, or add a new user affiliated with the Designer or the Administrator.

If you are changing the properties of an existing user account, you must pass a valid user ID number returned by the **GET USER LIST** command.

If the user account does not exist or has been deleted, the error -9979 is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**. Otherwise, you can call **Is user deleted** to test the user account before calling **Set user properties**.

User ID numbers can have the following values or ranges:

| User ID number | User description |
|---|---|
| 1 | Designer user |
| 2 | Administrator user |
| 3 to 15000 | User created by the Designer of the database (user #3 is the first user created by the Designer, user #4 the second, and so on). |
| -11 to -15000 | User created by the Administrator of the database (user #-11 is the first user created by the Administrator, user #-12 is the second, and so on). |

To add a new user affiliated with the Designer pass -1 in *userID*. To add a new user affiliated with the Administrator pass -2 in *userID*.
After the call, if the user is successfully added or modified, its unique ID number is returned in *userID*.

If you do not pass -1, -2 or a valid user ID number, **Set user properties** does nothing.

Before the call, you pass the new name, startup method, password, number of logins and date of last login for the user, in the *name*, *startup*, *password*, *nbLogin* and *lastLogin* parameters.You pass an unencrypted password in the *password* parameter. 4D will encrypt it for you before saving it in the user account.
If the new user name passed in *name* is not unique (there is already a user with the same name), the command does nothing and the error -9979 is returned. You can catch this error with an error-handling method installed using **ON ERR CALL**.

If you do not want to change all the properties of the user (aside from the memberships, see below), first call **GET USER PROPERTIES** and pass the returned values for the properties you want to leave unchanged.

If you do not want to change the password for an account, pass the * symbol as a value for the *password* parameter. This allows you to change the other properties of the user account without changing the password for the account.

If you do not pass the optional *memberships* parameter, the current memberships of the user are left unchanged. If you do not pass *memberships* when adding a user, the user will not belong to any group.
If you pass the optional *memberships* parameter, you change all the memberships for the user. Before the call, you must populate the *memberships* array with the unique ID numbers of the groups to which the user will belong.

If you pass the optional *groupOwner* parameter, you indicate the ID number of the user group “owner”, i.e. the default owner group of the objects created by this user.

The group ID numbers can be the following:

| Group ID number | Group description |
|---|---|
| 15001 to 32767 | Group created by the Designer or affiliated Group Owner<br>(group #15001 is the first group created by the Designer, group #15002 the second, and so on). |
| -15001 to -32768 | Group created by the Administrator or affiliated Group Owner<br>(group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on). |

To revoke all the memberships of a user, pass an empty *memberships* array.

## Error management

If you do not have the proper access privileges for calling **Set user properties** or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using **ON ERR CALL**.

# User in group

User in group ( user ; group ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| user | String | ⇒ | User name |
| group | String | ⇒ | Group name |
| Function result | Boolean | ⇐ | TRUE = user is in group FALSE = user is not in group |

## Description

**User in group** returns TRUE if *user* is in *group*.

## Example

The following example searches for specific invoices. If the current user is in the Executive group, he or she is allowed access to forms that display confidential information. If the user is not in the Executive group, a different form is displayed:

```
QUERY([Invoices];[Invoices]Retail>100)
If(User in group(Current user;"Executive"))
   FORM SET OUTPUT([Invoices];"Executive Output")
   FORM SET INPUT([Invoices];"Executive Input")
Else
   FORM SET OUTPUT([Invoices];"Standard Output")
   FORM SET INPUT([Invoices];"Standard Input")
End if
MODIFY SELECTION([Invoices];*)
```

# USERS TO BLOB

USERS TO BLOB ( users )

| Parameter | Type | | Description |
|---|---|---|---|
| users | BLOB | ⇨ | BLOB that must contain users |
| | | ⇦ | User accounts (encrypted) |

## Description

The **USERS TO BLOB** command stores in the BLOB *users* the list of all user accounts and database groups created by the Administrator.

Only the database Administrator or the Designer can execute this command. If another user tries to execute it, the command does nothing and a privilege error (-9949) is generated.

The generated BLOB is encrypted automatically and can only be read using the **BLOB TO USERS** command. You can store this BLOB in a file on your hard disk or in a field.

This command is the equivalent of recording groups and users from the Toolbar. The only difference is that it lets you store user accounts in a BLOB field and not just in a file.

This lets you keep a backup of users in the database data and, as such, implements a backup mechanism as well as a system to load users automatically when a database structure file is updated (information related to user accounts are stored by 4D in the database structure file).

# Validate password

Validate password ( userID ; password {; digest} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| userID | Longint, String | ⇒ | Unique user ID or User name |
| password | String | ⇒ | Unencrypted password |
| digest | Boolean | ⇒ | Digest password = True, Plain-text password (default) = False |
| Function result | Boolean | ⇐ | True = valid password False = invalid password |

## Description

**Validate password** returns True if the string passed in *password* is the password for the user account whose ID number or name is passed in *userID*.

The optional *digest* parameter indicates whether the *password* parameter contains a plain-text password or a hashed password (digest mode):

- When you pass **True**, this indicates that *password* contains a hashed password (digest mode),
- When you pass **False** or omit this parameter, this indicates that *password* contains a plain-text password.

This parameter is particularly helpful when using authentication database methods, in particular the **On 4D Mobile Authentication database method**.

The command execution is now delayed to prevent flooding (brute force attack), in other words, attempts of multiple user name/password combinations. As a result, after the 4th call to this command, it is run only after a period of 10 seconds. This delay is throughout the entire work station.

## Example 1

This example checks whether the password of the user “Hardy” is “Laurel”:

```
GET USER LIST(atUserName;alUserID)
$vlElem:=Find in array(atUserName;"Hardy")
If($vlElem>0)
   If(Validate password(alUserID{$vlElem};"Laurel"))
      ALERT("Yep!")
   Else
      ALERT("Too bad!")
   End if
Else
   ALERT("Unknown user name")
End if
```

## Example 2

In the **On 4D Mobile Authentication database method**, you want to test a connection request (using the 4D users of the database). You can just write:

```
$0:=Validate password($1;$2;$3)
```

# Variables

- CLEAR VARIABLE
- LOAD VARIABLES
- SAVE VARIABLES
- Undefined

# CLEAR VARIABLE

CLEAR VARIABLE ( variable )

| Parameter | Type | | Description |
|---|---|---|---|
| variable | Variable | ⇨ | Variable to clear |

## Description

**CLEAR VARIABLE** resets *variable* to its default type value (i.e., empty string for String and Text variables, *0* for numeric variables, no elements for arrays, etc.). The variable still exists in memory.

The variable you pass in *variable* can be a local, process or interprocess variable.

**Note:** You do not need to clear process variables when a process ends; 4D clears them automatically. Similarly, each local variable is automatically cleared when the method in which it was created completes execution.

## Example

In a form, you are using the drop-down list *asMyDropDown* whose sole purpose is user interface. In other words, you use that array during data entry, but once you are done with the form, you will no longer use that array. Consequently, during the On Unload event, you just get rid of the array:

```
   ` asMyDropDown drop-drop list object method
Case of
   :(Form event=On Load)
   ` Initialize the array one way or another
      ARRAY STRING(63;asMyDropDown;...)
   ` ...
   :(Form event=On Unload)
   ` No longer need the array
      CLEAR VARIABLE(asMyDropDown)
   ` ...
End case
```

# LOAD VARIABLES

LOAD VARIABLES ( document ; variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document containing 4D variables |
| variable | Variable | ⇒ | Variables to receive the values |

## Description

The **LOAD VARIABLES** command loads one or more variables from the document specified by *document*. The document must have been created using the **SAVE VARIABLES** command.

The variables *variable*, *variable2*...*variableN* are created; if they already exist, they are overwritten.

If you supply an empty string for *document*, the standard Open File dialog box appears, so the user can choose the document to open. If a document is chosen, the 4D system variable *Document* is set to the name of the document.

In compiled databases, each variable must be of the same type as those loaded from disk.

**WARNING:** This command does not support array variables. Use the new BLOB commands instead.

## Example

The following example loads three variables from a document named UserPrefs:

```
LOAD VARIABLES("User Prefs";vsName;vlCode;vgIconPicture)
```

## System variables and sets

If the variables are loaded properly, the **OK** system variable is set to 1; otherwise it is set to 0.

# SAVE VARIABLES

SAVE VARIABLES ( document ; variable {; variable2 ; ... ; variableN} )

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document in which to save the variables |
| variable | Variable | ⇒ | Variables to save |

## Description

The **SAVE VARIABLES** command saves one or more variables in the document whose name you pass in *document*.

The variables do not need to be of the same type, but must be of the String, Text, Real, Integer, Long Integer, Date, Time, Boolean, or Picture type.

If you pass an empty string for *document*, the standard Save File dialog box appears; the user can then choose the document to create. In this case, the 4D system variable *Document* is set to the name of the document if one is created.

If the variables are properly saved, the OK variable is set to 1. If not, OK is set to 0.

**Note:** When you write variables to documents with **SAVE VARIABLES**, 4D uses an internal data format. You can retrieve the variables only with the **LOAD VARIABLES** command. Do not use **RECEIVE PACKET** or **RECEIVE VARIABLE** to read a document created by **SAVE VARIABLES**.

**WARNING:** This command does not support array variables. Use the new BLOB commands instead.

## Example

The following example saves three variables to a document named UserPrefs:

```
SAVE VARIABLES("User Prefs";vsName;vlCode;vgIconPicture)
```

## System variables and sets

If the variables are saved properly, the **OK** system variable is set to 1; otherwise it is set to 0.

# Undefined

Undefined ( variable ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| variable | Variable | ⇒ | Variable to test |
| Function result | Boolean | ⇐ | True = Variable is currently undefined False = Variable is currently defined |

## Description

**Undefined** returns True if *variable* has not been defined, and False if *variable* has been defined. A variable is defined if it has been created via a compilation directive or if a value is assigned to it. It is undefined in all other cases.

If the database has been compiled, the **Undefined** function returns False for all variables.

## Example

The following code manages the creation of processes when a menu item for a particular module of your application is chosen. If the process already exists, you bring it to the front; if it does not exist, you start it. To do so, for each module of the application, you maintain an interprocess variable ◊*PID_...* that you initialize in the **On Startup database method**.

When developing the database, you add new modules. Instead modifying the **On Startup database method** (to add the initialization of the corresponding ◊*PID_...*) and then reopening the database to reinitialize everything each time you add a module, you use the **Undefined** command to manage the addition of the new module, on the fly:

```
 //M_ADD_CUSTOMERS global procedure

If(Undefined(◊PID_ADD_CUSTOMERS)) // Takes care of intermediate development stages
   C_LONGINT(◊PID_ADD_CUSTOMERS)
   ◊PID_ADD_CUSTOMERS:=0
End if

If(◊PID_ADD_CUSTOMERS=0)
   ◊PID_ADD_CUSTOMERS:=New process("P_ADD_CUSTOMERS";64*1024;"P_ADD_CUSTOMERS")
Else
   SHOW PROCESS(◊PID_ADD_CUSTOMERS)
   BRING TO FRONT(◊PID_ADD_CUSTOMERS)
End if
 //Note: P_ADD_CUSTOMERS, the process master method,
 // sets ◊PID_ADD_CUSTOMERS to zero when it ends.
```

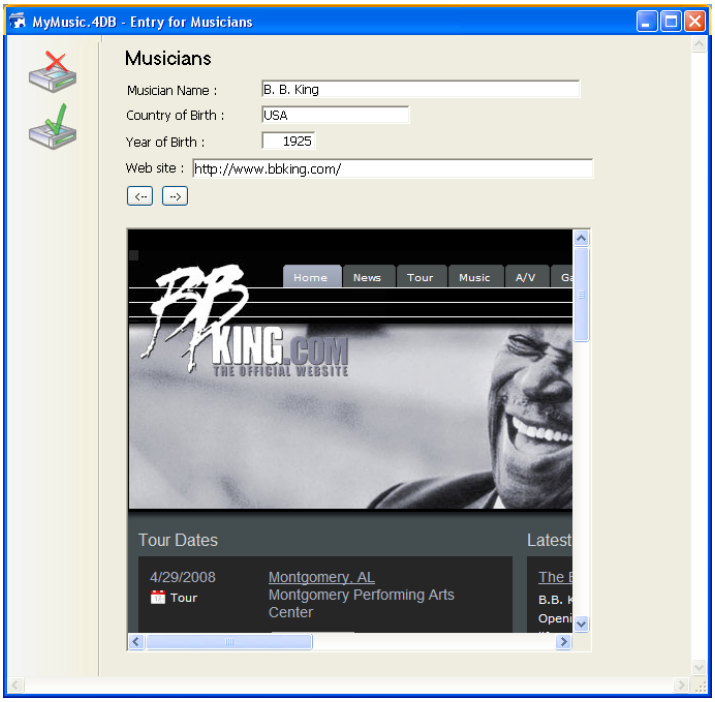
# Web Area

- Programmed management of Web Areas
- WA Back URL available
- WA Create URL history menu
- WA Evaluate JavaScript
- WA EXECUTE JAVASCRIPT FUNCTION
- WA Forward URL available
- WA Get current URL
- WA GET EXTERNAL LINKS FILTERS
- WA Get last filtered URL
- WA GET LAST URL ERROR
- WA Get page content
- WA Get page title
- WA GET PREFERENCE
- WA GET URL FILTERS
- WA GET URL HISTORY
- WA OPEN BACK URL
- WA OPEN FORWARD URL
- WA OPEN URL
- WA REFRESH CURRENT URL
- WA SET EXTERNAL LINKS FILTERS
- WA SET PAGE CONTENT
- WA SET PAGE TEXT LARGER
- WA SET PAGE TEXT SMALLER
- WA SET PREFERENCE
- WA SET URL FILTERS
- WA STOP LOADING URL

# Programmed management of Web Areas

The commands of this theme are dedicated to the programmed management of Web Area type form objects.

Web areas can display any type of Web content(*) within your 4D environment: HTML pages with static or dynamic contents, files, pictures, Javascript, etc. The following picture shows a Web area included in a form and displaying an HTML page:




(*) However, the use of Web plugins and Java applets is not recommended (see **Notes about use of Web areas**).

In addition to the commands of the Web Area theme, several standard actions and form events allow the developer to control the functioning of these Web areas. Specific variables can be used to exchange information between the area and the 4D environment. These tools can be used to develop a basic Web browser in your forms.

## Creating and addressing a Web area

A Web area is created using a variant of the Plug-in Area/Subform button found in the object bar of the 4D Form editor (for more information, please refer to **Web areas** in the *Design Reference* manual).

**Note:** When a Web area that uses the embedded Web rendering engine is displayed in a new process, in particular one created with the **New process** command, it is necessary to use the default value (0) for the *stack* parameter in order to ensure its correct display.

Like other dynamic form objects, a Web area has an object name and a variable name, which can be used to handle it by programming. The standard variable associated with a Web area object is of the Text type. More specifically, you can use the **OBJECT SET VISIBLE** and **OBJECT MOVE** commands with Web areas.

**Note**: The text variable associated with the Web area does not contain a reference therefore it cannot be passed as a parameter for a method. For example, for a Web area named *MyArea*, the following code cannot be used:

```
Mymethod(MyArea)
```

Code for Mymethod:

```
WA REFRESH CURRENT URL($1) //Does not work
```

For this type of programming, you will need to use pointers:

```
Mymethod(->MyArea)
```

Code for Mymethod:

```
WA REFRESH CURRENT URL($1->) //Works
```

## Management of associated variables

In addition to the standard object variable (see previous paragraph), two specific variables are automatically associated with each Web area:

- The "URL" variable
- The "Progression" variable.

You can name these variables as desired. These variables can be accessed in the Property List:

### URL Variable

"URL" is a String type variable. It contains the URL loaded or being loading by the associated Web area.
The association between the variable and the Web area works in both directions:

- If the user assigns a new URL to the variable, this URL is automatically loaded by the Web area.
- Any browsing done within the Web area will automatically update the contents of the variable.
  Schematically, this variable functions like the address area of a Web browser. You can represent it via a text area above the Web area.

**URL Variable and WA OPEN URL command**
The URL variable produces the same effects as the **WA OPEN URL** command. The following differences should nevertheless be noted:

- For access to documents, this variable only accepts URLs that are RFC-compliant ("file://c:/My%20Doc") and not system pathnames ("c:¥MyDoc"). The **WA OPEN URL** command accepts both notations.
- If the URL variable contains an empty string, the Web area does not attempt to load the URL. The **WA OPEN URL** command generates an error in this case.
- If the URL variable does not contain a protocol (http, mailto, file, etc.), the Web area adds "http://", which is not the case for the **WA OPEN URL** command.
- When the Web area is not displayed in the form (when it is located on another page of the form), executing the **WA OPEN URL** command has no effect, whereas assigning a value to the URL variable can be used to update the current URL.

### Progression Variable

"Progression" is a Longint type variable. It contains a value between 0 and 100, representing the percentage of loading that is complete for the page displayed in the Web area.
This variable is automatically updated by 4D. It is not possible to modify it manually.

## Accessing 4D methods

You can call 4D methods from the JavaScript code executed in a Web area and get values in return.

**Important:** This feature is only available if the Web area uses the embedded Web rendering engine.

### Configuring the Web area

To be able to call 4D methods from a Web area, you must check the **Access 4D methods** option for the area in the Property List:

**Note:** This option is only shown when **Use embedded Web rendering engine** option is checked.

When this property is checked, a special JavaScript object ($4d) is instantiated in the Web area, which you can use to manage calls to 4D project methods.

### Using the $4d object

When the **Access 4D methods** option is checked, the 4D embedded Web rendering engine supplies the area with a JavaScript object named *$4d* that you can associate with any 4D project method using the "." object notation.

For example, to call the ***HelloWorld*** 4D method, you just execute the following statement:

```
$4d.HelloWorld();
```

**Warning**: JavaScript is case sensitive so it is important to note that the object is named $4d (with a lowercase "d").

The syntax of calls to 4D methods is as follows:

```
$4d.4DMethodName(param1,paramN,function(result){})
```

- *param1...paramN*: You can pass as many parameters as you need to the 4D method.
  These parameters can be of any type supported by JavaScript (string, number, array, object).
- *function(result)*: Function to pass as last argument. This "callback" function is called synchronously once the 4D method finishes executing. It receives the *result* parameter:
  - *result*: Execution result of the 4D method, returned in the "$0" expression.
    This result can be of any type supported by JavaScript (string, number, array, object). You can use the **C_OBJECT** command to return the objects.
    **Note:** By default, 4D works in UTF-8. When you return text containing extended characters, for example characters with accents, make sure the encoding of the page displayed in the Web area is declared as UTF-8, otherwise the characters may be rendered incorrectly. In this case, add the following line in the HTML page to declare the encoding:
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

### Example 1

Given a 4D project method named ***today*** that does not receive parameters and returns the current date as a string.

4D code of ***today*** method:

```
C_TEXT($0)
$0:=String(Current date;System date long)
```

In the Web area, the 4D method can be called with the following syntax:

```
$4d.today()
```

The 4D method does not receive any parameters but it does return the value of *$0* to the callback function called by 4D after the execution of the method.
We want to display the date in the HTML page that is loaded by the Web area.

Here is the code of the HTML page:

```
<html> <head> <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" /> <script type="text/javascript">
$4d.today(function(dollarZero) {     var curDate = dollarZero;     document.getElementById("mydiv").innerHTML=curDate; });
</script> </head> <body>Today is: <div id="mydiv"></div> </body> </html>
```

### Example 2

The 4D project method ***calcSum*** receives parameters ($1...$n) and returns their sum in $0:

4D code of ***calcSum*** method:

```
C_REAL(${1}) // receives n REAL type parameters
C_REAL($0) // returns a Real
C_LONGINT($i;$n)
$n:=Count parameters
For($i;1;$n)
   $0:=$0+${$i}
End for
```

The JavaScript code run in the Web area is:

```
$4d.calcSum(33, 45, 75, 102.5, 7, function(dollarZero)     {         var result = dollarZero // result is 262.5     });
```

## Form events

Specific form events are intended for programmed management of Web areas, more particularly concerning the activation of links:

- On Begin URL Loading
- On URL Resource Loading

- On End URL Loading
- On URL Loading Error
- On URL Filtering
- On Open External Link
- On Window Opening Denied

In addition, Web areas support the following generic form events:

- On Load
- On Unload
- On Getting Focus
- On Losing Focus

For more information about these events, please refer to the description of the **Form event** command.

## Access to Web inspector

You can view and use a Web inspector within Web areas of your forms. The Web inspector is a debugger which is provided by the embedded Web engine. It allows to parse the code and the flow of information of the Web pages.

### Displaying the Web inspector

The following conditions must be met in order to view the Web inspector in a Web area:

- You must select the embedded Web rendering engine for the area (the Web inspector is only available with this configuration) (see **Use integrated Web rendering engine**)
- You must enable the context menu for the area (this menu is used to call the inspector) (see **Context Menu**)
- You must expressly enable the use of the inspector in the area by means of the following statement:

```
WA SET PREFERENCE(*;"WA";WA enable Web inspector;True)
```

  For more information, refer to the description of the **WA SET PREFERENCE** command.

### Using the Web inspector

When you have done the settings as described above, you then have new options such as **Inspect Element** in the context menu of the area:

When you select this option, the Web inspector window is displayed.

The Web inspector is included in the embedded Web rendering engine. For a detailed description of the features of this debugger, refer to the documentation provided by the Web rendering engine.

## Notes about use of Web areas

### User interface

When the form is executed, standard browser interface functions are available to the user in the Web area, which permit interaction with other form areas:

- **Edit menu commands:** When the Web area has the focus, the **Edit** menu commands can be used to carry out actions such as copy, paste, select all, etc., according to the selection.
- **Context menu:** It is possible to use the standard context menu of the system with the Web area (see **Context Menu**). Display of the context menu can be controlled using the **WA SET PREFERENCE** command.
- **Drag and drop:** The user can drag and drop text, pictures and documents within the Web area or between a Web area and the 4D form objects, according to the 4D object properties.
  For security reasons, changing the contents of a Web area by means of dragging and dropping a file or URL is not allowed by default beginning with 4D v14 R2. In this case, the mouse cursor displays a "forbidden" icon  . You have to use the **WA SET PREFERENCE** command to explicitly allow the dropping of URLs or files in the area.

### Web Area and Web server conflict (Windows)

Under Windows, it is not recommended to access, via a Web area, the Web server of the 4D application containing the area because this configuration could lead to a conflict that freezes the application. Of course, a remote 4D can access the Web

server of 4D Server, but not its own Web server.

### Web plugins and Java applets

The use of Web plugins and Java applets is **not recommended** in Web areas because they may lead to instability in the operation of 4D, particularly at the event management level.

### Insertion of protocol (Mac OS)

The URLs handled by programming in Web areas under Mac OS must begin with the protocol. For example, you need to pass the string "http://www.mysite.com" and not just "www.mysite.com".

# WA Back URL available

WA Back URL available ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | Boolean | ⇐ | True if there is a previous URL in the sequence of URLs opened;otherwise, False |

## Description

The **WA Back URL available** command finds out whether there is a previous URL available in the sequence of URLs opened in the Web area designated by the * and *object* parameters.

The command returns **True** if a URL exists and **False** otherwise. More particularly, this command can be used, in a custom interface, to enable or disable navigation buttons.

# WA Create URL history menu

WA Create URL history menu ( {* ;} object {; direction} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| direction | Longint | ⇒ | 0 or omitted=List of previous URLs, 1=List of next URLs |
| Function result | MenuRef | ⇐ | Menu reference |

## Description

The **WA Create URL history menu** command creates and fills a menu that can be used directly for navigation among the URLs visited during the session in the Web area designated by the * and *object* parameters. It can be used to build a custom navigation interface.

The information provided concerns the session; in other words, the navigation carried out in the same Web area as long as the form has not been closed.

Pass a value indicating the list to recover in *direction*. You can use one of the following constants, found in the **Web Area** theme:

| Constant | Type | Value |
|---|---|---|
| WA next URLs | Longint | 1 |
| WA previous URLs | Longint | 0 |

If you omit the *direction* parameter, the value 0 is used.

Once the menu has been generated, you can display it using the 4D **Dynamic pop up menu** command and you can work with it using the standard 4D menu management commands. The string returned by this command contains the URL of the page visited (see example).

Call the **RELEASE MENU** command to delete a URL history menu when it is no longer useful.

## Example

The following code can be associated with a 3D button having a pop-up menu entitled "Previous":

```
Case of
 //Single click
   :(Form event=On Clicked)
      WA OPEN BACK URL(WA_area)
 //Click on arrow -> display of pop up
   :(Form event=On Alternative Click)
 //Create a previous history menu
      $Menu:=WA Create URL history menu(WA_area;WA previous URLs)
 //Display this menu in a pop-up
      $URL:=Dynamic pop up menu($Menu)
 //If an item is selected
      If($URL#"")
 //Open Web page
         WA OPEN URL(WA_area;$URL)
      End if
 //Delete menu to free up memory
      RELEASE MENU($Menu)
End case
```

# WA Evaluate JavaScript

WA Evaluate JavaScript ( {* ;} object ; jsCode {; type} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or Variable (if * is omitted) |
| jsCode | String | ➡ | JavaScript code |
| type | Longint | ➡ | Type into which to convert result |
| Function result | Date, Object, Pointer, Real, Text, Time | ↩ | Result of evaluation |

## Description

The **WA Evaluate JavaScript** command executes, in the Web area designated by the * and *object* parameters, the JavaScript code passed in *jsCode* and returns the result.

By default, the command returns values as strings. You can use the optional *type* parameter to specify typing for the value returned. To do this, pass one of the following constants, found in the "**Field and Variable Types**" theme:

| Constant | Type | Value |
|---|---|---|
| Is Boolean | Longint | 6 |
| Is date | Longint | 4 |
| Is longint | Longint | 9 |
| Is object | Longint | 38 |
| Is real | Longint | 1 |
| Is text | Longint | 2 |
| Is time | Longint | 11 |

## Example 1

This example of JavaScript code causes the previous URL to be displayed:

```
$result:=WA Evaluate JavaScript(MyWArea;"history.back()")
```

## Example 2

This example shows a few evaluations with conversion of the values received.

JavaScript functions placed in an HTML file:

```
<!DOCTYPE html> <html>     <head>         <script>         function evalLong(){             return 123;         }
function evalText(){            return "456";        }        function evalObject(){            return {a:1,b:"hello
world"};         }        function evalDate(){            return new Date();         }     </script>     </head>
<body>         TEST PAGE     </body> </html>
```

In the 4D form method, you write:

```
If(Form event=On Load)
   WA OPEN URL(*;"Web Area";"C:¥¥myDatabase¥¥index.html")
End if
```

You can then evaluate the JavaScript code from 4D:

```
$Eval1:=WA Evaluate JavaScript(*;"Web Area";"evalLong()";Is longint)
 //$Eval1 = 123
 //$Eval1 = "123" if type is omitted
$Eval2:=WA Evaluate JavaScript(*;"Web Area";"evalText()";Is text)
 //$Eval2 = "456"
$Eval3:=WA Evaluate JavaScript(*;"Web Area";"evalObject()";Is object)
```

```
 //$Eval3 = {"a":1,"b":"hello world"}
$Eval4:=WA Evaluate JavaScript(*;"Web Area";"evalDate()";ls date)
 // $Eval4 = 06/21/13
 // $Eval4 = "2013-06-21T14:45:09.694Z" if type is omitted
```

# WA EXECUTE JAVASCRIPT FUNCTION

WA EXECUTE JAVASCRIPT FUNCTION ( {* ;} object ; jsFunction ; result|* {; param}{; param2 ; ... ; paramN} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| jsFunction | String | ⇒ | Name of JavaScript function to execute |
| result\|* | Variable | ⇒ | * for a function with no result or |
| | | ⇐ | Function result (if expected) |
| param | String, Number, Date, Object | ⇒ | Parameter(s) to pass to function |

## Description

The **WA EXECUTE JAVASCRIPT FUNCTION** command executes, in the Web area designated by the * and *object* parameters, the JavaScript function *jsFunction* and optionally returns its result in the *result* parameter.

If the function does not return a result, pass * in the *result* parameter.

You can pass one or more parameters containing the parameters of the function in *param*.

The command supports several types of parameters for both input (*param*) and output (*result*). You can pass and retrieve data of the number, date, object and string types.

## Example 1

Calling a JavaScript function with 3 parameters:

```
$JavaScriptFunction:="TheFunctionToBeExecuted"
$Param1:="10"
$Param2:="true"
$Param3:="1,000.2" `note "," as thousands separator and "." as the decimal separator

WA EXECUTE JAVASCRIPT FUNCTION(MyWArea;$JavaScriptFunction;$Result;$Param1;$Param2;$Param3)
```

## Example 2

The "getCustomerInfo" JavaScript function receive a number ID as parameter and returns an object:

```
C_OBJECT($Result)
C_LONGINT($ID)
$ID:=1000
WA EXECUTE JAVASCRIPT FUNCTION(*;"WA";"getCustomerInfo";$Result;$ID)
```

# WA Forward URL available

WA Forward URL available ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | Boolean | ⇐ | True if there is a following URL in the sequence of URLs opened;otherwise, False |

## Description

The **WA Forward URL available** command finds out whether there is a following URL available in the sequence of URLs opened in the Web area designated by the * and *object* parameters.

The command returns **True** if a URL exists and **False** otherwise. More particularly, this command can be used, in a custom interface, to enable or disable navigation buttons.

# WA Get current URL

WA Get current URL ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | String | ⇐ | URL currently loaded in the Web area |

## Description

The **WA Get current URL** command returns the URL address of the page displayed in the Web area designated by the * and *object* parameters.

If the current URL is not available, the command returns an empty string.

If the Web page is completely loaded, the value returned by the function is the same as that of the "URL" variable associated with the Web area. If the page is in the process of being loaded, the two values will be different: the function returns the completely loaded URL and the variable contains the URL in the process of being loaded.

## Example

The page displayed is the URL "www.apple.com" and the "www.4d.com" page is in the process of being loaded:

```
$url:=WA Get current URL(MyWArea) `returns "http://www.apple.com"
 `The associated URL variable contains "http://www.4d.com"
```

# WA GET EXTERNAL LINKS FILTERS

WA GET EXTERNAL LINKS FILTERS ( {* ;} object ; filtersArr ; allowDenyArr )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| filtersArr | String array | ⇐ | Filters array |
| allowDenyArr | Boolean array | ⇐ | Allow-deny array |

## Description

The **WA GET EXTERNAL LINKS FILTERS** command returns, in the *filtersArr* and *allowDenyArr* arrays, the external link filters of the Web area designated by the * and *object* parameters. If no filter is active, the arrays are returned empty.

The filters are installed by the **WA SET EXTERNAL LINKS FILTERS** command. If the arrays are reinitialized during the session, the **WA GET EXTERNAL LINKS FILTERS** command can be used to find out the current settings.

# WA Get last filtered URL

WA Get last filtered URL ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | String | ⇐ | Last filtered URL |

## Description

The **WA Get last filtered URL** command returns the last URL that was filtered in the Web area designated by the * and *object* parameters.

The URL may have been filtered for one of the following reasons:

- The URL was denied because of a filter (**WA SET URL FILTERS** command),
- The link is open in the default browser (**WA SET EXTERNAL LINKS FILTERS** command),
- The URL attempts to open a pop-up window.

It is advisable to call this command in the context of the On URL Filtering, On Open External Link and On Window Opening Denied form events in order to find out the URL that was filtered.

# WA GET LAST URL ERROR

WA GET LAST URL ERROR ( {* ;} object ; url ; description ; errorCode )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| url | String | ⇐ | URL at origin of error |
| description | String | ⇐ | Description of error (Mac OS) |
| errorCode | Longint | ⇐ | Error code |

## Description

The **WA GET LAST URL ERROR** command recovers several items of information about the last error that occurred in the Web area designated by the * and *object* parameters.

This information is returned in three variables:

- *url*: URL causing error.
- *description* (Mac OS only): A text describing the error (if available). If it is not possible to associate a text with the error, an empty string is returned. Under Windows, this parameter is always returned empty.
- *errorCode*: The error code.
  - If the code is >=400, it is an error related to the HTTP protocol. For more information about this type of error, refer to the following address:
    *http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html*
  - Otherwise, it is an error returned by the WebKit (Mac OS) or ActiveX (Windows).

It is recommended to call this command within the framework of the On URL Loading Error form event to find out the cause of the error that just occurred.

# WA Get page content

WA Get page content ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | String | ⇐ | HTML source code |

## Description

The **WA Get page content** command returns the HTML code of the current page or the page being displayed in the Web area designated by the * and *object* parameters.

This command returns an empty string if the contents of the current page is not available.

# WA Get page title

WA Get page title ( {* ;} object ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| Function result | String | ⇐ | Title of current page |

## Description

The **WA Get page title** command returns the title of the current page or the page being displayed in the Web area designated by the * and *object* parameters. The title corresponds to the HTML "Title" tag.

This command returns an empty string if there is no title available for the current URL.

# WA GET PREFERENCE

WA GET PREFERENCE ( {* ;} object ; selector ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| selector | Longint | ⇒ | Preference to get |
| value | Variable | ⇐ | Current value of the preference |

## Description

The **WA GET PREFERENCE** command gets the current value of the preference in the Web area designated by the * and *object* parameters.

Pass the preference whose value you want to get in the *selector* parameter. You can pass one of the following constants, found in the **Web Area** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| WA enable contextual menu | Longint | 4 | Allow the display of a standard contextual menu in the Web area |
| WA enable Java applets | Longint | 1 | Allow the execution of Java applets in the Web area. |
| WA enable JavaScript | Longint | 2 | Allow the execution of JavaScript code in the Web area |
| WA enable plugins | Longint | 3 | Allow the installation of plug-ins in the Web area |
| WA enable URL drop | Longint | 101 | Allow dropping of URLs or files in the Web area (default = False) |
| WA enable Web inspector | Longint | 100 | Allow the display of the Web inspector in the area |

For more information about these preferences, refer to the description of the **WA SET PREFERENCE** command.

In the *value* parameter, pass a variable that will receive the current value of the preference. The type of variable depends on the preference. The *value* variable is always a Boolean: it contains **True** if the preference is active and **False** otherwise.

# WA GET URL FILTERS

WA GET URL FILTERS ( {* ;} object ; filtersArr ; allowDenyArr )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| filtersArr | String array | ⇐ | Filters array |
| allowDenyArr | Boolean array | ⇐ | Allow-deny array |

## Description

The **WA GET URL FILTERS** command returns, in the *filtersArr* and *allowDenyArr* arrays, the filters that are active in the Web area designated by the * and *object* parameters. If no filter is active, the arrays are returned empty.

The filters are installed by the **WA SET URL FILTERS** command. If the arrays are reinitialized during the session, the **WA GET URL FILTERS** command can be used to find out the current settings

# WA GET URL HISTORY

WA GET URL HISTORY ( {* ;} object ; urlsArr {; direction {; titlesArr}} )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| urlsArr | String array | ⇐ | Array of URLs visited |
| direction | Longint | ⇒ | 0 or omitted=List of previous URLs, 1=List of next URLs |
| titlesArr | String array | ⇐ | Array of window titles |

## Description

The **WA GET URL HISTORY** command returns one or two arrays containing the URLs visited during the session in the Web area designated by the * and *object* parameters. It can be used to build a custom navigation interface.

The information provided concerns the session; in other words, the navigation carried out in the same Web area as long as the form has not been closed.

The *urlsArr* array is filled with the list of URLs visited. Depending on the value of the *direction* parameter (if it is passed), the array recovers the list of previous URLs (default operation), or the list of next URLs. These lists correspond to the content of the standard Back and Forward buttons of browsers.

The URLs are classed by chronological order.

Pass a value indicating the list to recover in *direction*. You can use one of the following constants, found in the **Web Area** theme:

| Constant | Type | Value |
|---|---|---|
| WA next URLs | Longint | 1 |
| WA previous URLs | Longint | 0 |

If you omit the *direction* parameter, the value 0 is used.

If it is passed, the *titlesArr* parameter contains the list of window names associated with the URLs. This array is synchronized with the *urlsArr* array.

# WA OPEN BACK URL

WA OPEN BACK URL ( {* ;} object )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |

## Description

The **WA OPEN BACK URL** command loads the previous URL in the sequence of URLs opened into the Web area designated by the * and *object* parameters.

If there is no previous URL, the command does nothing. You can test whether a previous URL is available using the **WA Back URL available** command.

# WA OPEN FORWARD URL

WA OPEN FORWARD URL ( {* ;} object )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |

## Description

The **WA OPEN FORWARD URL** command loads the next URL in the sequence of URLs opened into the Web area designated by the * and *object* parameters.

If there is no next URL (in other words, if the user has never returned to a previous URL), the command does nothing. You can test whether a next URL is available using the **WA Forward URL available** command.

# WA OPEN URL

WA OPEN URL ( {* ;} object ; url )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or Variable (if * is omitted) |
| url | String | ➡ | URL to load in Web area |

## Description

The **WA OPEN URL** command loads the URL passed in the *url* parameter into the Web area designated by the * and *object* parameters.

If an empty string is passed in *url*, the command does nothing and no error is generated. To load a blank page into the Web area, pass the string "about:blank" in *url*.

Like the existing **OPEN URL** command, **WA OPEN URL** accepts several types of syntaxes in the *url* parameter to designate the files:

- posix syntax: "file:///c:/My%20File"
- system syntax: "c:¥MyFolder¥MyFile" (Windows) or "MyDisk:MyFolder:MyFile" (Mac OS).

**Note:** For compatibility, the "file://" syntax (using two "/") is accepted in 4D but it does not conform to RFC. We recommend using the "file:///" syntax (with three "/") that complies with RFC.

On Mac OS, when FileVault is activated, you must use the Posix syntax. You can transform paths of the system using the **Convert path system to POSIX** command.

This command has the same effect as modifying the value of the "URL" variable associated with the area. For example, if the variable of the area is named MyWArea_url:

```
MyWArea_url:="http://www.4d.com/"
```

is the same as:

```
WA OPEN URL(MyWArea;"http://www.4d.com/")
```

# WA REFRESH CURRENT URL

WA REFRESH CURRENT URL ( {* ;} object )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |

## Description

The **WA REFRESH CURRENT URL** command reloads the current URL displayed in the Web area designated by the * and *object* parameters.

# WA SET EXTERNAL LINKS FILTERS

WA SET EXTERNAL LINKS FILTERS ( {* ;} object ; filtersArr ; allowDenyArr )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| filtersArr | String array | ⇒ | Filters array |
| allowDenyArr | Boolean array | ⇒ | Allow-deny array |

## Description

The **WA SET EXTERNAL LINKS FILTERS** command sets up one or more external link filters for the Web area designated by the * and *object* parameters. External link filters determine whether a URL associated with the current page via a link must be opened in the Web area or in the default Web browser of the machine.

When the user clicks on a link in the current page, 4D consults the list of external link filters in order to check whether the URL requested must be opened in the browser of the machine. If so, the page corresponding to the URL is displayed in the Web browser and the On Open External Link form event is generated. Otherwise (default operation), the page corresponding to the URL is displayed in the Web area. The evaluation of the URL is based on the contents of the *filtersArr* and *allowDenyArr* arrays.

The *filtersArr* and *allowDenyArr* arrays must be synchronized.

- Each element of the *filtersArr* array must contain a URL to be filtered. You can use the * as a wildcard to replace one or more characters.
- Each corresponding element in the *allowDenyArr* array must contain a Boolean indicating whether the URL must be opened in the Web area (**True**) or in the Web browser (**False**).

If there is a contradiction at the configuration level (the same URL is both allowed and denied), the last setting is the one taken into account.

To disable URL filtering, call the command and pass empty arrays or pass, respectively, the values "*" and **True** in the last elements of the *filtersArr* and *allowDenyArr* arrays.

**Important:** The filtering established by the **WA SET URL FILTERS** command is taken into account before that of the **WA SET EXTERNAL LINKS FILTERS** command. This means that if a URL is denied because of a **WA SET URL FILTERS** command filter, it cannot be opened in the browser even if it is explicitly specified by the **WA SET EXTERNAL LINKS FILTERS** command (see example 2).

## Example 1

This example causes sites to be opened in external browsers:

```
ARRAY STRING(0;$filters;0)
ARRAY BOOLEAN($AllowDeny;0)

APPEND TO ARRAY($filters;"*www.google.*") `Select "google"
APPEND TO ARRAY($AllowDeny;False)
 `False: this link will be opened in an external browser
APPEND TO ARRAY($filters;"*www.apple.*")
APPEND TO ARRAY($AllowDeny;False)
 `False: this link will be opened in an external browser
WA SET EXTERNAL LINKS FILTERS(MyWArea;$filters;$AllowDeny)
```

## Example 2

This example combines the filtering of both sites and external links:

```
ARRAY STRING(0;$filters;0)
ARRAY BOOLEAN($AllowDeny;0)
APPEND TO ARRAY($filters;"*www.google.*") `Select "google"
```

```
APPEND TO ARRAY($AllowDeny;False) `Deny this link
WA SET URL FILTERS(MyWArea;$filters;$AllowDeny)

ARRAY STRING(0;$filters;0)
ARRAY BOOLEAN($AllowDeny;0)
APPEND TO ARRAY($filters;"*www.google.*") `Select "google"
APPEND TO ARRAY($AllowDeny;False)
 `False: this link should be opened in an external browser but this setting
 `has no effect because the link will be blocked by the URL filtering.
WA SET EXTERNAL LINKS FILTERS(MyWArea;$filters;$AllowDeny)
```

# WA SET PAGE CONTENT

WA SET PAGE CONTENT ( {* ;} object ; content ; baseURL )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| content | String | ⇒ | HTML source code |
| baseURL | String | ⇒ | URL for relative references (Mac OS) |

## Description

The **WA SET PAGE CONTENT** command replaces the page displayed in the Web area designated by the * and *object* parameters by the HTML code passed in the *content* parameter.

The *baseURL* parameter specifies, under Mac OS, a base URL that will be added in front of any relative links found in the page.

Under Windows, this parameter has no effect and the base URL is not specified so it is not possible to use relative references on this platform.

**Note:** Under Windows, you cannot call this command unless a page has already been loaded previously into the Web area. If necessary, you can pass the "about:blank" URL in order to load a blank page.

## Example

Displays "Hello world!" and sets a "file:///" base URL (Mac OS only):

```
WA SET PAGE CONTENT(MyWArea;"<html><body><h1>Hello World!</h1></body></html>";"file:///")
```

# WA SET PAGE TEXT LARGER

WA SET PAGE TEXT LARGER ( {* ;} object )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |

## Description

The **WA SET PAGE TEXT LARGER** command increases the size of the text displayed in the Web area designated by the * and *object* parameters.

Under Mac OS, the scope of this command is the 4D session: the configuration carried out by this command is not retained after the 4D application is closed.

Under Windows, the scope of this command is global: the configuration is retained after the 4D application is closed.

# WA SET PAGE TEXT SMALLER

WA SET PAGE TEXT SMALLER ( {* ;} object )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |

## Description

The **WA SET PAGE TEXT SMALLER** command reduces the size of the text displayed in the Web area designated by the * and *object* parameters.

Under Mac OS, the scope of this command is the 4D session: the configuration carried out by this command is not retained after the 4D application is closed.

Under Windows, the scope of this command is global: the configuration is retained after the 4D application is closed.

# WA SET PREFERENCE

WA SET PREFERENCE ( {* ;} object ; selector ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ➡ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ➡ | Object name (if * is specified) or Variable (if * is omitted) |
| selector | Longint | ➡ | Preference to be modified |
| value | Boolean | ➡ | Value of the preference (True = allowed, False = not allowed) |

## Description

The **WA SET PREFERENCE** command sets different preferences for the Web area designated by the * and *object* parameters.

Pass the preference to be modified in the *selector* parameter and the value to be assigned to it in the *value* parameter. In *selector*, you can pass one of the following constants, found in the **Web Area** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| WA enable contextual menu | Longint | 4 | Allow the display of a standard contextual menu in the Web area |
| WA enable Java applets | Longint | 1 | Allow the execution of Java applets in the Web area. |
| WA enable JavaScript | Longint | 2 | Allow the execution of JavaScript code in the Web area |
| WA enable plugins | Longint | 3 | Allow the installation of plug-ins in the Web area |
| WA enable URL drop | Longint | 101 | Allow dropping of URLs or files in the Web area (default = False) |
| WA enable Web inspector | Longint | 100 | Allow the display of the Web inspector in the area |

For each preference, pass **True** in *value* to activate it and **False** to deactivate it.

**Note:** The use of Web plugins and Java applets is **not recommended** in Web areas because they may lead to instability in the operation of 4D, particularly at the event management level.

## Example

To enable URL drops in the 'myarea' Web area:

```
WA SET PREFERENCE(*;"myarea";WA enable URL drop;True)
```

# WA SET URL FILTERS

WA SET URL FILTERS ( {* ;} object ; filtersArr ; allowDenyArr )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |
| filtersArr | String array | ⇒ | Filters array |
| allowDenyArr | Boolean array | ⇒ | Allow-deny array |

## Description

The **WA SET URL FILTERS** command sets up one or more filters for the Web area designated by the * and *object* parameters.

Before loading any page requested by the user, 4D consults the list of filters in order to check whether or not the target URL is allowed. The evaluation of the URL is based on the contents of the *filtersArr* and *allowDenyArr* arrays.

If the requested URL is not allowed, it is not loaded and the On URL Filtering form event is generated.

The *filtersArr* and *allowDenyArr* arrays must be synchronized.

- Each element of the *filtersArr* array must contain a URL to be filtered. You can use the * as a wildcard to replace one or more characters.
- Each corresponding element in the *allowDenyArr* array must contain a Boolean indicating whether the URL must be allowed (**True**) or denied (**False**).

If there is a contradiction at the configuration level (the same URL is both allowed and denied), the last setting is the one taken into account.

To disable URL filtering, call the command and pass empty arrays or pass, respectively, the values "*" and **True** in the last elements of the *filtersArr* and *allowDenyArr* arrays.

Once the command has been executed, the filters become a property of the Web area. If the *filtersArr* and *allowDenyArr* arrays are deleted or reinitialized, the filters remain active as long as the command has not been executed again. To find out the active filters for an area, you must use the **WA GET URL FILTERS** command.

**Important:** The filtering of URLs carried out by this command only applies to the "URL" variable associated with the Web area (variable usually enterable and displayed in the form).
The filtering does not apply to the **WA OPEN URL** command, nor to the other navigation commands.

## Example 1

You want to deny access for all the .org, .net and .fr Web sites:

```
ARRAY TEXT($filters;0)
ARRAY BOOLEAN($AllowDeny;0)

APPEND TO ARRAY($filters;"*.org")
APPEND TO ARRAY($AllowDeny;False)
APPEND TO ARRAY($filters;"*.net")
APPEND TO ARRAY($AllowDeny;False)
APPEND TO ARRAY($filters;"*.fr")
APPEND TO ARRAY($AllowDeny;False)
WA SET URL FILTERS(MyWArea;$filters;$AllowDeny)
```

## Example 2

You want to deny access for all Web sites except Russian ones (.ru):

```
ARRAY TEXT($filters;0)
ARRAY BOOLEAN($AllowDeny;0)

APPEND TO ARRAY($filters;"*") `Select all
```

```
APPEND TO ARRAY($AllowDeny;False) `Deny all
APPEND TO ARRAY($filters;"www.*.ru") `Select *.ru
APPEND TO ARRAY($AllowDeny;True) `Allow
WA SET URL FILTERS(MyWArea;$filters;$AllowDeny)
```

## Example 3

You want to allow access only to 4D Web sites (.com, .fr, .es, etc.):

```
ARRAY TEXT($filters;0)
ARRAY BOOLEAN($AllowDeny;0)

APPEND TO ARRAY($filters;"*") `Select all
APPEND TO ARRAY($AllowDeny;False) `Deny all
APPEND TO ARRAY($filters;"www.4D.*") `Select 4d.fr, 4d.com...
APPEND TO ARRAY($AllowDeny;True) `Allow
WA SET URL FILTERS(MyWArea;$filters;$AllowDeny)
```

## Example 4

You want to allow local access to the documentation only (found in the folder C://doc):

```
ARRAY TEXT($filters;0)
ARRAY BOOLEAN($AllowDeny;0)

APPEND TO ARRAY($filters;"*") `Select all
APPEND TO ARRAY($AllowDeny;False) `Deny all
APPEND TO ARRAY($filters;"file://C:/doc/*")
 `Select the path file:// allowed
APPEND TO ARRAY($AllowDeny;True) `Allow
WA SET URL FILTERS(MyWArea;$filters;$AllowDeny)
```

## Example 5

You want to allow access for all sites except one, for example the Elcaro site:

```
ARRAY TEXT($filters;0)
ARRAY BOOLEAN($AllowDeny;0)

APPEND TO ARRAY($filters;"*")
APPEND TO ARRAY($AllowDeny;True) `Allow all
APPEND TO ARRAY($filters;"*elcaro*") `Deny all that contain elcaro
APPEND TO ARRAY($AllowDeny;False)
WA SET URL FILTERS(MyWArea;$filters;$AllowDeny)
```

## Example 6

You want to deny access to specific IP addresses:

```
ARRAY TEXT($filters;0)
ARRAY BOOLEAN($AllowDeny;0)

APPEND TO ARRAY($filters;"*") `Select all
APPEND TO ARRAY($AllowDeny;True) `Allow all
APPEND TO ARRAY($filters;86.83.*")  `Select IP addresses beginning with 86.83.
APPEND TO ARRAY($AllowDeny;False) `Deny
APPEND TO ARRAY($filters;86.1*")  `Select IP addresses beginning with 86.1 (86.10, 86.135 etc.)
APPEND TO ARRAY($AllowDeny;False) `Deny
WA SET URL FILTERS(MyWArea;$filters;$AllowDeny)
 `(Note that the IP address of a domain may vary).
```

# WA STOP LOADING URL

WA STOP LOADING URL ( {* ;} object )

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, object is an object name (string) If omitted, object is a variable |
| object | Form object | ⇒ | Object name (if * is specified) or Variable (if * is omitted) |

## Description

The **WA STOP LOADING URL** command stops loading the resources of the current URL of the Web area designated by the * and *object* parameters.

# Web Server

- Web Server, Overview
- Web server configuration and connection management
- Support of IPv6
- Connection Security
- On Web Authentication Database Method
- On Web Connection Database Method
- On Web Close Process database method
- Web Sessions Management
- Semi-dynamic pages
- URLs and Form Actions
- Binding 4D objects with HTML objects
- Web Server Settings
- Using preemptive Web processes
- Information about the Web Site
- Using TLS Protocol
- XML and WML Support
- WEB CLOSE SESSION
- WEB GET BODY PART
- WEB Get body part count
- WEB Get Current Session ID
- WEB GET HTTP BODY
- WEB GET HTTP HEADER
- WEB GET OPTION
- WEB GET SESSION EXPIRATION
- WEB Get session process count
- WEB GET STATISTICS
- WEB GET VARIABLES
- WEB Is secured connection
- WEB Is server running
- WEB SEND BLOB
- WEB SEND FILE
- WEB SEND HTTP REDIRECT
- WEB SEND RAW DATA
- WEB SEND TEXT
- WEB SET HOME PAGE
- WEB SET HTTP HEADER
- WEB SET OPTION
- WEB SET ROOT FOLDER
- WEB START SERVER
- WEB STOP SERVER
- WEB Validate digest
- *_o_SET CGI EXECUTABLE*
- *_o_SET WEB DISPLAY LIMITS*
- *_o_SET WEB TIMEOUT*
- *_o_Web Context*

# Web Server, Overview

---

4D in local mode, 4D in remote mode and 4D Server include a Web Server engine that enables you to publish 4D databases or any type of HTML page on the Web. The principal characteristics of the 4D Web Server engine are:

- **Easy publication**

You can start or stop publication of the database on the Web at any time. To do so, you just need to choose a menu command or execute a language command.

- **Dedicated database methods**

**On Web Authentication Database Method** and **On Web Connection Database Method** are the entry points of requests in the Web server; they can be used to evaluate and route any type of request.

- **Use of special tags and URLs**

The 4D Web server offers numerous mechanisms that enable interaction with user actions, in particular:
- special tags can be included in Web pages in order to initiate processing by the Web server at the time when they are sent to browsers.
- special URLs that enable 4D to be called in order to execute any action.
- these URLs can also be used as form actions to trigger processing when the user posts HTML forms.

- **Managing user sessions**

The 4D Web server includes complete automatic features for easily managing Web sessions (user sessions) based on cookies.

- **Access Security**

Several automatic configuration options allow you to grant specific access authorizations to Web browsers or to use the password system integrated into 4D. You can define a "Generic Web User" to simplify access management within the database.
The **On Web Authentication Database Method** allows you to evaluate any request before it is processed by the Web server. Moreover, the ability to define a default HTML root folder allows you to restrict access to files on disk.
Finally, you must designate individually the project methods that may be executed via the Web.

- **SSL Connections**

Your 4D Web server can communicate with browsers in secured mode through the SSL protocol (Secured Socket Layer). This protocol, compatible with most Web browsers, authenticates the sender and receiver and guaranties the confidentiality and integrity of the exchanged information.

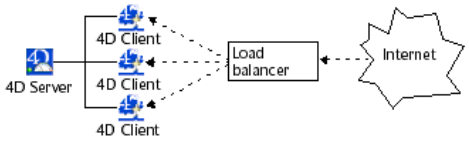
- **Extended support for Internet formats**

The 4D Web server is HTTP/1.1 compatible and supports XML documents and WML (Wireless Markup Language) technology. The 4D Web server also extends the support of gzip compression: after a "negotiation" between the Web server and client, all exchanges can potentially be compressed, for an immediate performance boost.

- **Simultaneous operation of databases**
  - **4D in local mode and the Web**
    If you publish a 4D database on the Web using 4D in local mode, you can simultaneously:
    - Use the database locally with 4D
    - Connect to the database using Web browsers.
  - **4D Server and the Web**
    If you publish a 4D database on the Web using 4D Server, you can simultaneously connect to and operate the 4D database, using:
    - 4D remote workstations
    - Web browsers.

- **4D in remote mode and the Web**
  When a 4D database is published on the Web with 4D client, it is possible to connect to the 4D database and to simultaneously use it:
  - via 4D remote machines
  - via Web browsers. In this case, if the database is also published with 4D Server, the Web browsers can connect to the published database via a 4D client machine or via 4D Server. Moreover, this allows different data access modes to be handled (public, administration, etc.).
  The basic mechanisms of the 4D Web server are used in a similar manner by 4D in remote mode. The operation of language commands is usually identical, whether the command be executed on 4D in local mode, 4D Server or 4D in remote mode. The main point is that commands are applied to the Web site of the machine on which they are executed. You must manage this using the **Execute on server** / **EXECUTE ON CLIENT** commands.

- **Load balancing with 4D clients**: since any 4D machine in remote mode can be used as a Web server, you can set up a dynamic Web server system with a load balancer. This offers extensive development possibilities, including, more particularly:
  - the setting-up of a load-balancing system in order to optimize the performance of the 4D Web server: using a mirror of the Web site that is installed on each 4D Web server, a load balancer (hardware or software) will send requests to the client machines on the basis of their current load.




  - the setting-up of a fault tolerance Web server: the 4D Web site is mirrored on two or more 4D client machines. If one 4D Web server fails, another one takes over.
  - the creation of different views of the same data, for instance depending on the origin of the requests. Within a company network, a Web server on a protected 4D client machine can serve Intranet requests and a Web server on another 4D client machine, located beyond the firewall, will serve Internet requests.
  - the distribution of tasks between Web servers on different 4D client machines: one 4D Web server can be in charge of SOAP requests, another can handle standard requests, and so on.

# Web server configuration and connection management

4D and 4D Server include a Web server (also called the HTTP server) that enables you to publish the data of your databases on the Web, transparently and dynamically.

This section describes the steps necessary for launching the Web server and for the connection of browsers, as well as the process of connection management.

## Conditions for publishing a 4D database on the Web

To be able to launch the HTTP server of 4D or 4D Server, you must have the elements described below:

- A "4D Web Application" license. For more information, please refer to your 4D installation guide.
- Web connections are made over the network using the TCP/IP protocol. Consequently:
  - You must have TCP/IP installed on your machine and correctly configured. Refer to your computer or Operating System manuals for more information.
  - If you want to use SSL for network connections, make sure that requested components are correctly installed (see the **Using TLS Protocol** section).
- After all the previous points have been checked and taken care of, you need to start the Web server from within 4D. This last point is discussed further on in this section.

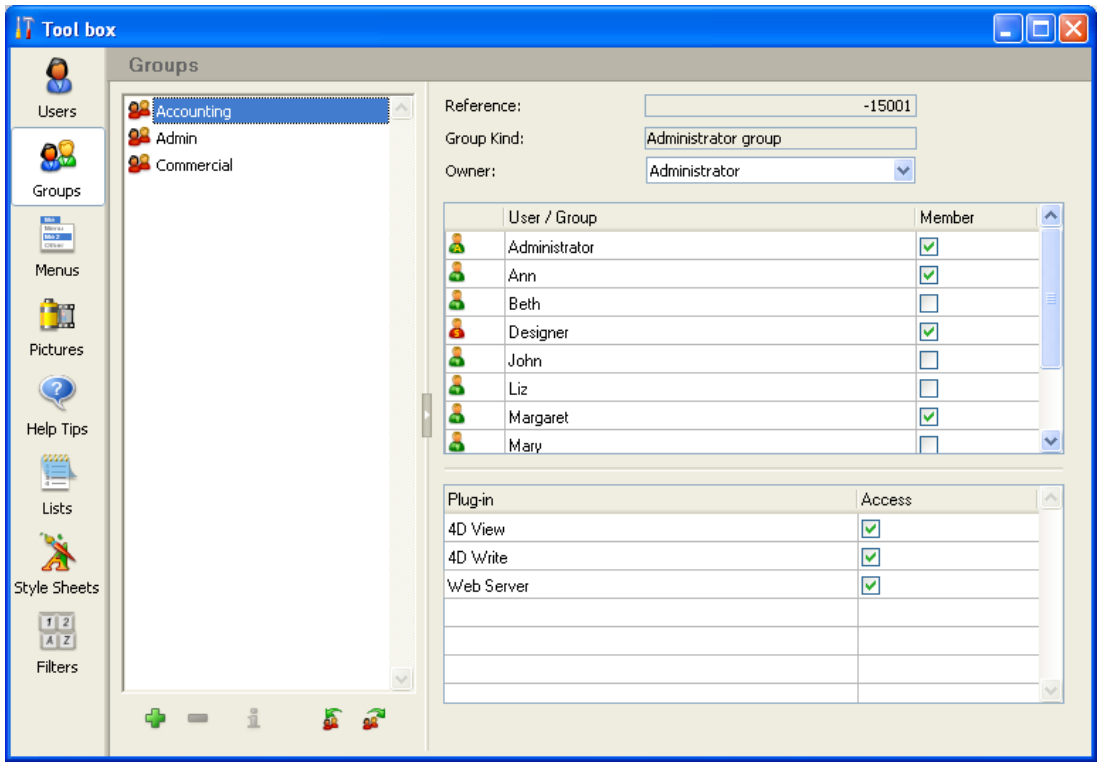
### Publication authorization (4D in remote mode)

By default, any 4D client machine can publish the database to which it is connected on the Web. However, you can control the possibility of Web publication for each remote 4D machine by using the 4D password system.

In fact, 4D Web licenses are considered as plug-in licenses by 4D Server. Therefore, in the same way as for plug-ins, you must retrict the right to use Web Server licenses to a specific group of users.

To do this, display the **Groups** page in the Toolbox using 4D (you must have suitable access authorization to modify these parameters).

Select a group in the list on the left, then check the **Access** option next to the **4D Client Web Server** line in the Plug-in distribution area:




Above: only users belonging to the "Web" group are authorized to publish their 4D machine as a Web server.

## HelperTool under Mac OS X

Under Mac OS X, using TCP/IP ports reserved for Web publishing (ports 0 to 1023) requires specific access privileges. In order for you to be able to use these ports, 4D provides a utility program named HelperTool. When this program is installed, it retrieves the appropriate access rights and automatically takes charge of opening the Web ports. This mechanism functions with 4D (all modes), 4D Server and 4D Volume Desktop executable applications.

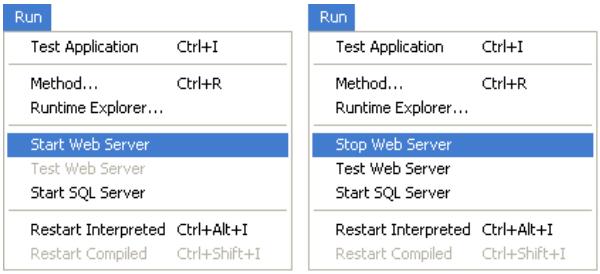
The HelperTool application is included in the 4D software. Installation takes place automatically during the first opening of a port <1024 on the machine. The user is informed that a tool is going to be installed and is prompted to enter a name and an administrator password for the machine. This operation only takes place once.
The application is renamed "com.4D.HelperTool" and is installed in the folder "/Library/PrivilegedHelperTools/." After the initial sequence, the 4D Web server can be started and stopped transparently, regardless of the 4D version used.

## Starting the 4D Web Server

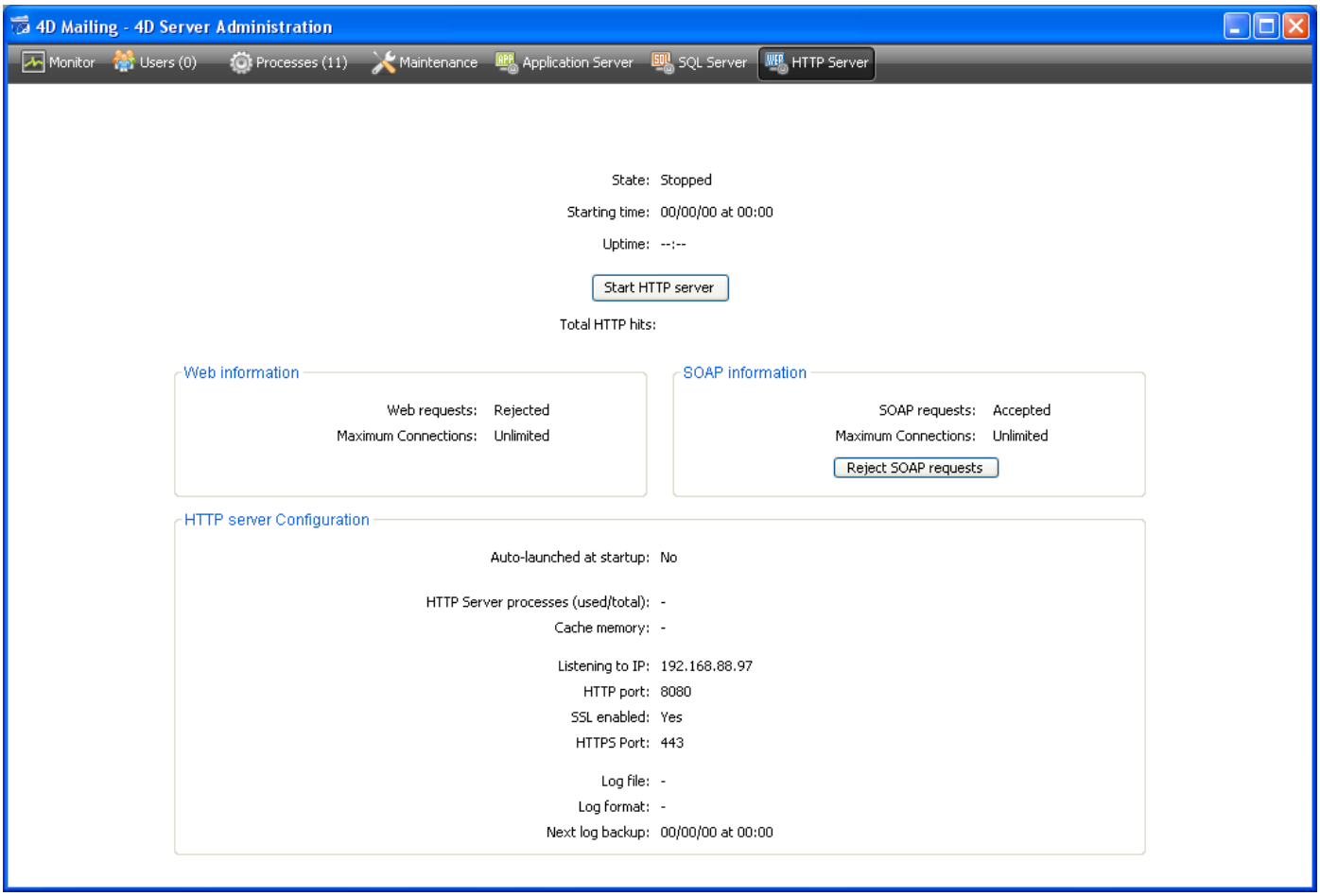
The 4D WebServer can be started in three different ways:

- Using the **Run** menu of 4D or the HTTP Server page of 4D Server(**Start HTTP server** button). These commands allow you to start and stop the WebServer at your convenience:

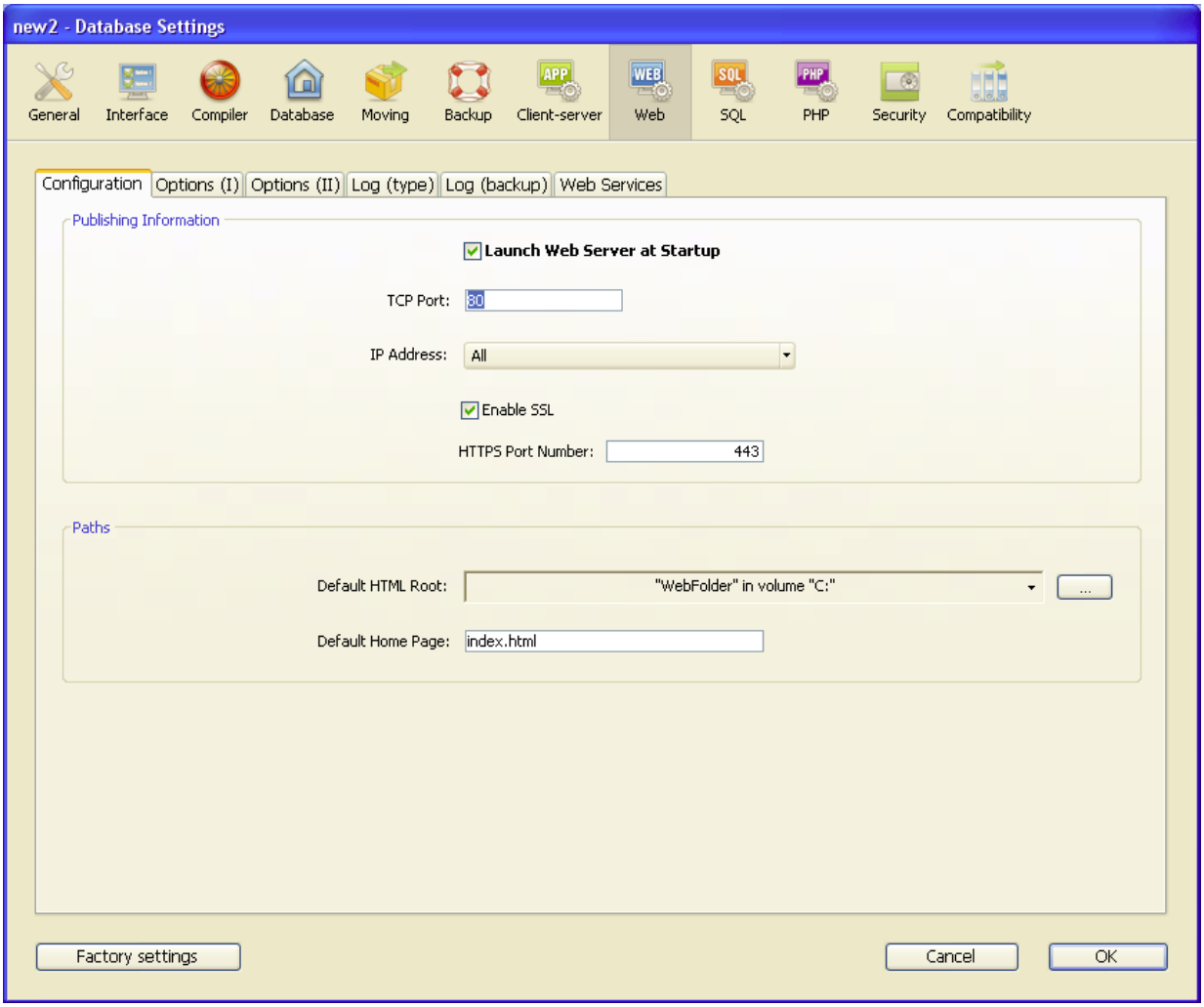
4D:




4D Server:




- Automatically starting it each time the 4D application is opened. To do this, display the **Configuration** page of the **Web** theme of the Database Settings:



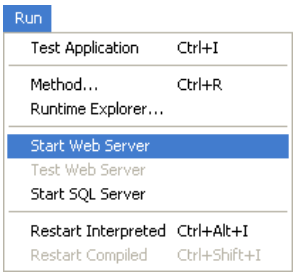

In the “Publishing Information” section, select the **Launch Web Server at Startup** check box, then click **OK**. Once this is done, the database will be automatically published on the Web each time you open it with 4D or 4D Server.

- Programmatically, by calling the **WEB START SERVER** command.

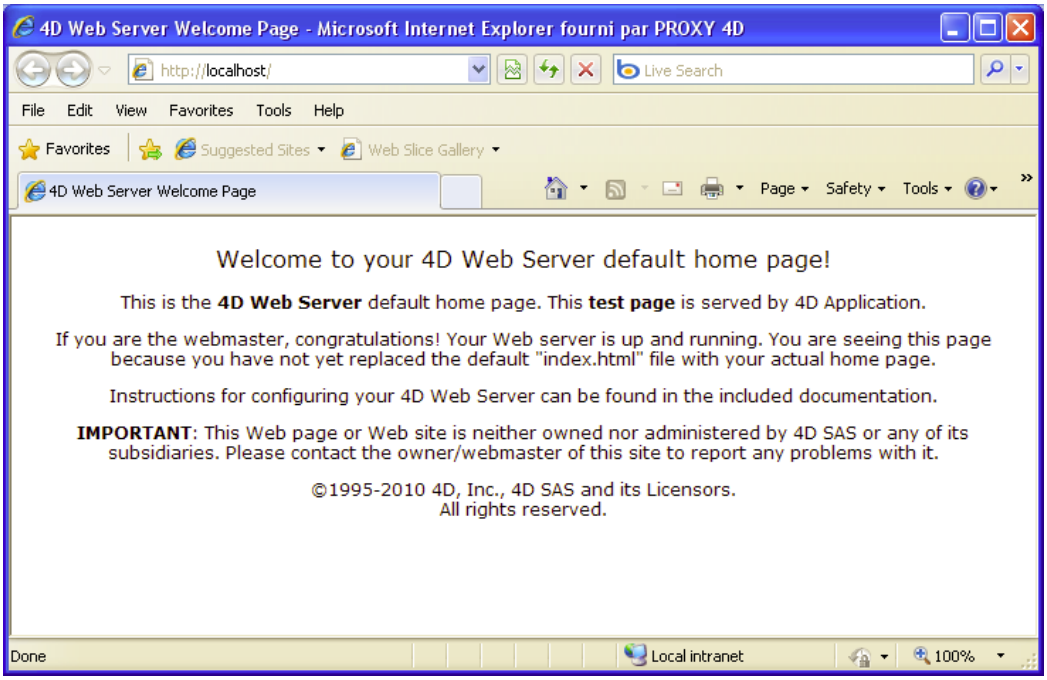
**Tip:** You do not need to quit 4D and reopen your database to start or stop publishing a database on the Web. You can interrupt and restart the Web server as many times as you want, using the **Run** menu, the **Start HTTP server** button or by calling the **WEB START SERVER** and **WEB STOP SERVER** commands.

## Testing the Web server

The **Test Web Server** command can be used to make sure the built-in Web server is functioning correctly (4D only). This command is accessible in the **Run** menu when the Web server is launched:




When you select this command, the home page of the Web site published by the 4D application is displayed in a window of your default Web browser:




This command lets you verify that the Web server, home page display, etc. work correctly. The page is called using the URL Localhost, which is the standard shortcut designating the IP address of the machine on which the Web browser is executed. The command takes into account the TCP publication port number specified in the database settings.

## Connecting to a 4D database published on the Web

After you have started publishing a 4D database on the Web, you can connect to it using a Web browser. To do so:

- If your Web site has a registered name (i.e., " *www.flowersforever.com*"), indicate that name in the Open, Address, or Location area of your browser. Then press **Enter** to connect.
- If your Web Site does not have a registered name, indicate the IP address of your machine (i.e., 123.4.567.89) in the Open, Address, or Location area of your browser. Then press **Enter**.

At this time, your browser should display the home page of your Web site. If you have published a database in keeping with standard configurations, you should obtain the default home page of the 4D Web server. This page lets you test the connection and the server operation.

You may also encounter one of the following situations:

1. The connection fails and you get a message such as "...the server may not be accepting connections or may be busy...". In this case, check the following:

- Verify that the name or the IP address you entered is correct.
- Verify that 4D or 4D Server is up and running and has started its Web server.
- Check if the database is configured for being served on a TCP Port other than the default Web TCP Port (see situation 3).
- Check whether TCP/IP is correctly configured on both the server and browser machines. Both machines must be on the same net and subnet, or your routers must be correctly configured.
- Check your hardware connections.
- If you are not locally testing your own site, but rather attempting to connect to a Web database served on Internet or Intranet by someone else, ultimately, the message might be true: the server may be off or busy. So, retry later until you can log on, or contact the Web provider.

2. You connect, but you get an HTTP 404 "File not found" error. This means that the site home page has not be served. In this case, check that the home page actually exists at the location defined in the database settings (see the **Web Server Settings** section) or using the **WEB SET HOME PAGE** command.

3. You connect, but you do NOT obtain the Web page you were expecting! This can occur when you have several Web servers running simultaneously on the same machine. Examples:

- You are running only one 4D Web database on a Windows system that is already running its own Web server.
- You are running several 4D Web databases on the same machine.

In this kind of situation, you need to change the TCP port number on which your 4D Web database is published. To do so, refer to the **Web Server Settings** section.

**Note:** If your database is protected by a password system, you may have to enter a valid user name and password (for more information, refer to the **Connection Security** section).

## Web Processes

Each time a Web browser attempts to connect to the database, the request is handled as follows:

- First, one or more temporary local 4D processes called **Web Connection Processes** are created in order to evaluate and manage the connection with the Web browser.
  These temporary processes manage every HTTP request. They execute quickly and are then aborted or delayed. In order to optimize the Web server, once a request has been handled, 4D freezes this “pool” of Web processes for a few seconds so that it can reuse them if necessary when another request arrives. You can customize this behavior (timeout, minimum and maximum number of processes to be kept in the "pool") using the **SET DATABASE PARAMETER** command.
- The Web process handles the processing of the request and sends a response (if necessary) to the browser. The temporary process is then aborted or delayed (see above).

## Support of IPv6

---

Starting with version 14, 4D supports IPv6 address notation. This concerns the following 4D integrated servers:

- the Web server as well as the SOAP server,
- the SQL server.

**Note:** For more information about IPv6, refer to the following specification: RFC 2460.

Support of IPv6 is transparent for users and for 4D developers: the program accepts either IPv6 or IPv4 connections indiscriminately when the listening "IP address" of the server is set to **All** (see **Defining the IP Address for the HTTP Requests** (HTTP server) and **SQL Server Publishing Preferences** (SQL server).

However, you should pay attention to the following points:

- **Indication of port numbers**
  Since IPv6 notation uses colons (:), adding port numbers may lead to some confusion, for example:

  ```
  2001:0DB8::85a3:0:ac1f:8001 // IPv6 address
  2001:0DB8::85a3:0:ac1f:8001:8081 // IPv6 address with port 8081
  ```

  To avoid this confusion, we recommend using the [ ] notation whenever you combine an IPv6 address with a port number, for instance:

  ```
  [2001:0DB8::85a3:0:ac1f:8001]:8081 //IPv6 address with port 8081
  ```

- **No longer any warning when TCP port is occupied**
  Unlike previous versions of 4D, when the server is set to respond on "all" IP addresses with 4D v14, if the TCP port is being used by another application, this is no longer indicated when the server is started. In fact, 4D server does not detect any error in this case because the port remains free on the IPv6 address. However, it is not possible to access it using the IPv4 address of the machine, nor by means of the local address: 127.0.0.1.
  If your 4D server does not seem to be responding on the port defined, you can test the address [::1] on the server machine (equivalent to 127.0.0.1 for IPv6, add :*portNum* to test another port number). If 4D responds, it is likely that another application is using the port in IPv4.

- **IPv4-mapped IPv6 addresses**
  To standardize processing, 4D provides a standard hybrid representation of IPv4 addresses in IPv6. These addresses are written with a 96-bit prefix in IPv6 format, followed by 32 bits written in the dot-decimal notation of IPv4. For example, ::ffff:192.168.2.34 represents the IPv4 address 192.168.2.34.

## Connection Security

The security of your 4D Web Server is based on the following elements:

- The combination of the Web password management system (BASIC mode or DIGEST mode) and the **On Web Authentication Database Method**,
- The definition of a Generic Web User,
- The definition of a HTML Root folder by default,
- The definition of the “Available through 4D tags and URLs (4DACTION…)” property for each project method of the database.
- The option for the specific support of synchronization requests through HTTP.

**Note:** The security of the connection itself can be managed through the SSL protocol. For more information, refer to the **Using TLS Protocol** section.

### Password Management System for Web Access

#### BASIC Mode and DIGEST Mode

In the Database Settings, you can set the access control system that you want to apply to your Web server. Two authentication modes are provided: BASIC mode and DIGEST mode. The authentication mode concerns the way the information concerning the user name and password are collected and processed.

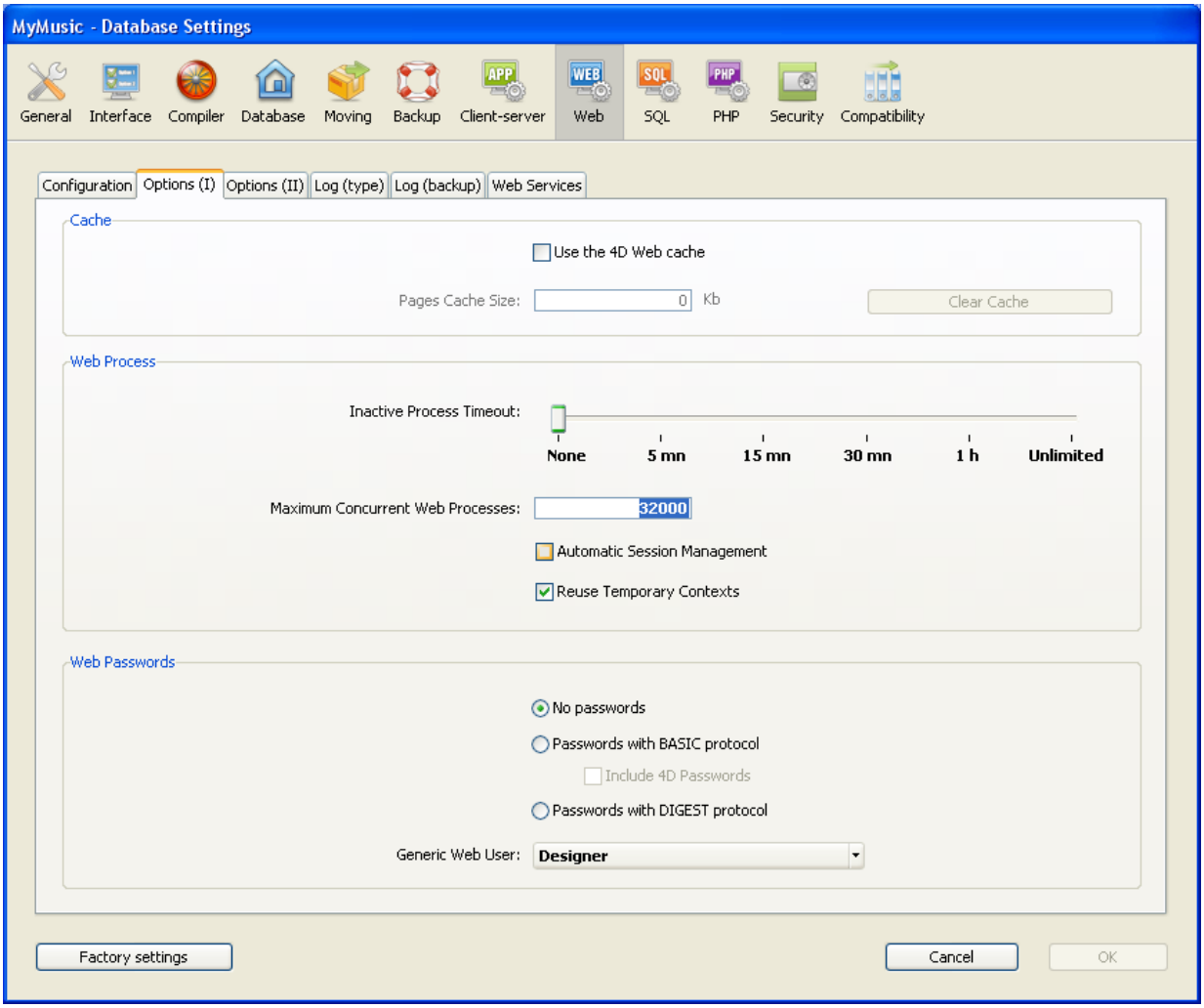
- In BASIC mode, the name and password entered by the user are sent unencrypted in the HTTP requests. This does not ensure total system security since this information could be intercepted and used by a third party.
- The DIGEST mode provides a greater level of security since the authentication information is processed by a one-way process called hashing which makes their contents impossible to decipher.

For the user, the use of either authentication mode is transparent.

**Notes:**

- For compatibility reasons, the BASIC authentication mode is used by default in 4D databases that are converted to version 11 (if the “Use Passwords” option was checked in the previous version). You must explicitly activate the Digest mode.
- Digest authentication is an HTTP1.1 function and is not supported by all browsers. For example, only versions 5.0 and later of Microsoft Internet Explorer accept this mode. If a browser that does not support this functionality sends a request to a Web server when Digest authentication is activated, the server will reject the request and return an error message to the browser.

You can now define, in the Database Settings dialog box, the access control system you want to apply to your Web server. To do this, choose the **Options (I)** page of the **Web** theme:




In the "Passwords" area, three options are available to you:

- **No passwords**: No authentication is carried out for connections to the Web server. In this case:
  - If the **On Web Authentication Database Method** exists, it is executed and, in addition to $1 and $2, only the IP addresses of the browser and the server ($3 and $4) are provided, the user name and password ($5 and $6) are empty. In this case, you can filter connections according to the IP address of the browser and/or the requested IP address of the server.
  - If the **On Web Authentication Database Method** does not exist, connections are automatically accepted.
- **Passwords with BASIC protocol**: Standard authentication in BASIC mode. When a user connects to the server, a dialog box appears on their browser in order for them to enter their user name and password. These two values are then sent to the **On Web Authentication Database Method** along with the other connection parameters (IP address and port, URL...) so that you can process them.
  This mode provides access to the **Include 4D passwords** option that allows you to use, instead of or in addition to your own password system, 4D's database password system (as defined in 4D).
- **Passwords with DIGEST protocol**: Authentication in DIGEST mode. As in BASIC mode, users must enter their name and password when they connect. These two values are then sent encrypted to the **On Web Authentication Database Method** with the other connection parameters. You must authenticate a user using the **WEB Validate digest** command.

**Notes:**

- You must restart the Web server in order for the changes made to these parameters to be taken into account
- With the 4D Client Web server, keep in mind that all the sites published by the 4D Client machines will share the same table of users. Validation of users/passwords is carried out by the 4D Server application.

### BASIC Mode: Combination of passwords and the On Web Authentication Database Method

If you use the BASIC mode, the system that filters connections to the 4D Web server depends on the combination of two parameters:

- The Web password options in the Database Settings dialog box,
- The existence of the **On Web Authentication Database Method**.

Here are the different resulting systems:

The “Passwords with BASIC protocol” option is selected and the “Include 4D Passwords” option is not selected.

- If the **On Web Authentication Database Method** exists, it is executed and all its parameters are given. You can therefore filter more precisely the connections according to the user name, password, and/or the browser’s or Web server’s IP address.
- If the **On Web Authentication Database Method** doesn’t exist, the connection is automatically refused and a message indicating that the Authentication method doesn’t exist is sent to the browser.

**Note:** If the user name sent by the browser is an empty string and if the **On Web Authentication Database Method** doesn’t exist, a password dialog box is sent to the browser.

The “Passwords with BASIC protocol” and “Include 4D Passwords” options are selected.

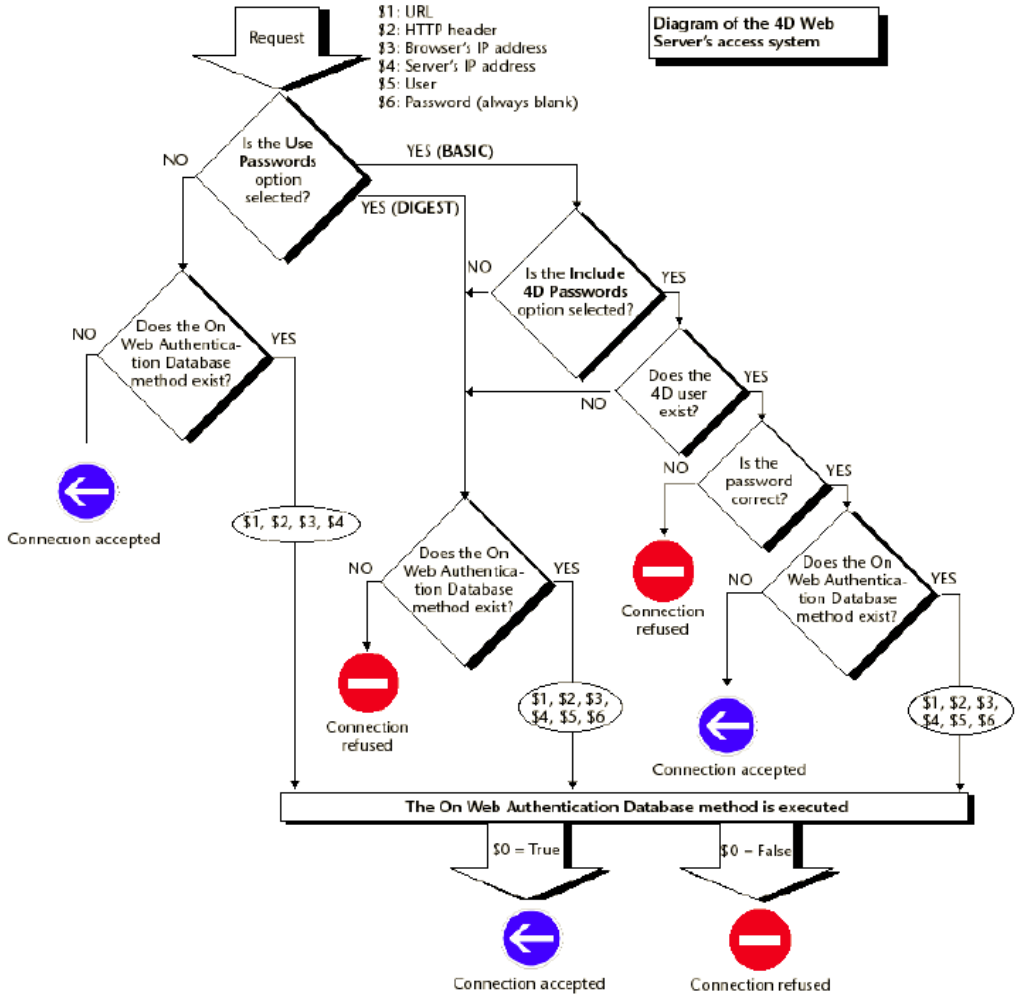
- If the user name sent by the browser exists in the table of 4D users and the password is correct, the connection is accepted. If the password is incorrect, the connection is refused.
- If the user name sent by the browser doesn’t exist in 4D, two results are then possible:
  - If the **On Web Authentication Database Method** exists, the parameters $1, $2, $3, $4, $5, and $6 are returned. You can therefore filter the connections according to the user name, password, and/or the browser’s or Web server’s IP address.
  - If the **On Web Authentication Database Method** doesn’t exist, the connection is refused.

### DIGEST Mode

Unlike BASIC mode, the DIGEST mode is not compatible with standard 4D passwords: it is not possible to use 4D passwords as Web IDs. The “Include 4D passwords” option is dimmed when this mode is selected. The IDs for Web users must be managed in a customized manner (for example, via a table).

When the DIGEST mode is activated, the $6 parameter (password) is always returned empty in the **On Web Authentication Database Method**. In fact, when using this mode, this information does not pass by the network as clear text (unencrypted). It is therefore imperative in this case to evaluate connection requests using the **WEB Validate digest** command.

The operation of the 4D Web server's access system is summarized in the following diagram:




### About robots (security note)

Certain robots (query engines, spiders...) scroll through Web servers and static pages. If you want robots to be able to access your entire site, you can define which URLs they are not allowed to access.
To do so, put the ROBOTS.TXT file at the server's root. This file must be structured in the following manner:

```
User-Agent: <name>
Disallow: <URL> or <beginning of the URL>
```

For example:

```
User-Agent: *
Disallow: /4D
Disallow: /%23%23
Disallow: /GIFS/
```

"User-Agent: *" means that all robots are affected.
"Disallow: /4D" means that robots are not allowed to access URLs beginning with /4D.
"Disallow: /%23%23" means that robots are not allowed to access URLs beginning with /%23%23.
"Disallow: /GIFS/' means that robots are not allowed to access the /GIFS/ folder or its subfolders.

Another example:

```
User-Agent: *
Disallow: /
```

In this case, robots are not allowed to access the entire site.

## Generic Web User

You can designate a user, previously defined in the 4D password table, as a "Generic Web User." In this case, each browser that connects to the database can use the access authorizations and restrictions associated with this generic user. You can therefore easily control the browser's access to the different parts of the database.

**Note:** Do not confuse this option, which allows you to restrict the browser's access to different parts of the application (methods, forms, etc.), with the Web server's connection control system, managed by the password system and the **On Web Authentication Database Method**.

To define a Generic Web User:

1. In the Design mode, create at least one user with the Users editor of the Tool Box.
   You can associate a password with the user if you wish.
2. In the different 4D editors, authorize or restrict access to this user.
3. In the Database Settings dialog, choose the **Options (I)** page of the **Web** theme.
   The "Web Passwords" area contains the **Generic Web User** drop-down list.By default, the Generic Web User is the Designer and the browsers have full access to the entire database.
4. Choose a user in the drop-down list and validate the dialog box

All the Web browsers that are authorized to connect to the database will benefit from the access authorizations and restrictions associated with this Generic Web User (except when the BASIC mode and the "Include 4D Passwords" option are checked and the user that connects does not exist in the 4D password table, see below).

### Interaction with the BASIC protocol

The "Passwords with BASIC protocol" option does not influence how the Generic Web User operates. Whatever the state of this option, the access authorizations and restrictions associated with the "Generic Web User" will be applied to all the Web browsers that are authorized to connect to the database.

However, when the "Include 4D passwords" option is selected, two possible results can occur:

- The user's name and password don't exist in 4D's password table. In this case, if the connection has been accepted by the **On Web Authentication Database Method**, the Generic Web User's access rights will be applied to the browser.
- If the user's name and password exist in 4D's password table, the "Generic Web User" parameter is ignored. The user connects with his own access rights.

## Default HTML Root

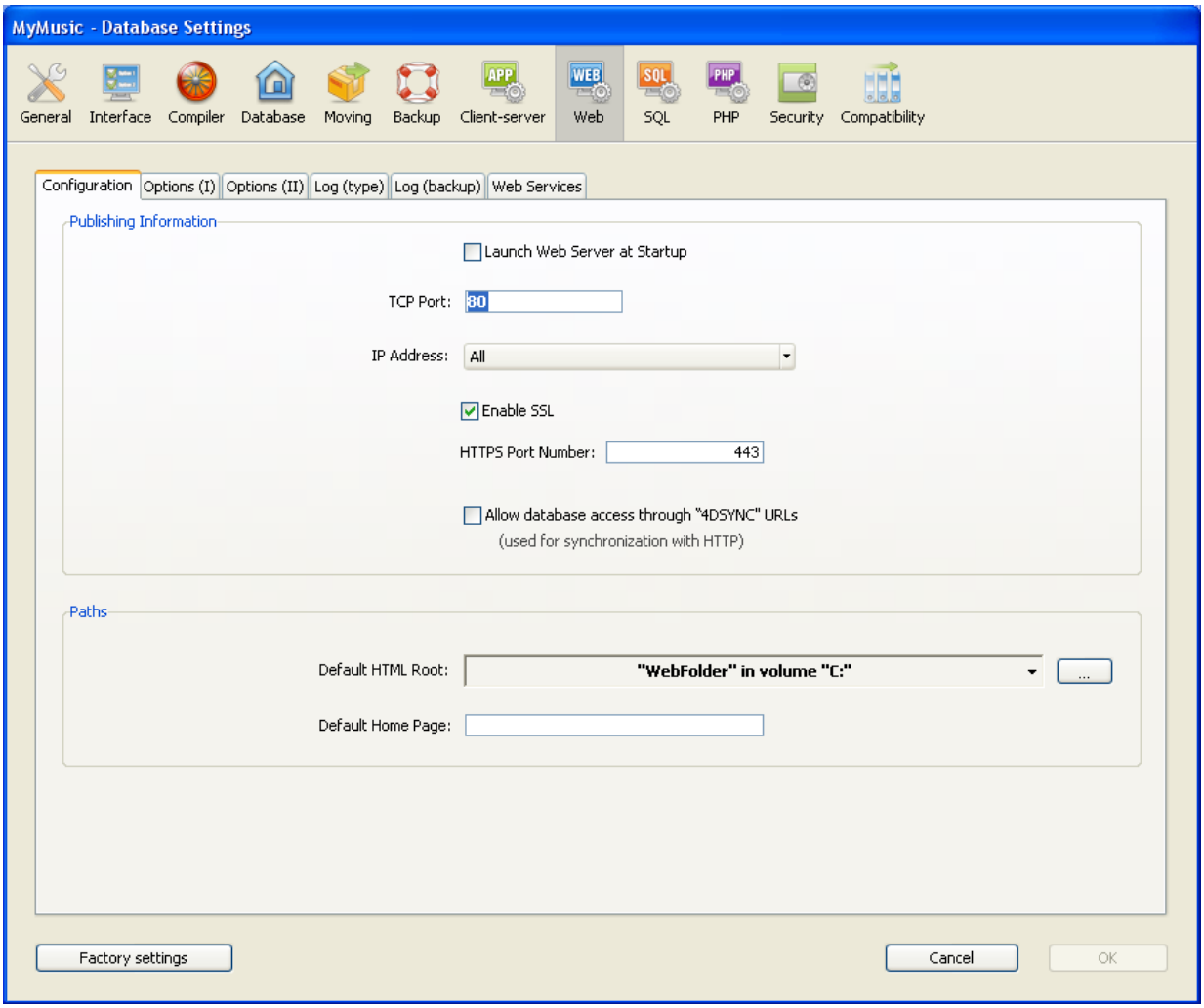
This option in the Database Settings allows you to define the folder in which 4D will search for the static and semi-dynamic HTML pages, pictures, etc., to send to the browsers.

Moreover, the HTML root folder defines, on the Web server hard drive, the hierarchical level above which the files will not be accessible. This access restriction applies to URLs sent to Web browsers as well as to 4D's Web server commands, such as **WEB SEND FILE**. If a URL is sent to the database by a browser or if a 4D command tries to access a file located above the HTML root folder, an error is returned indicating that the file has not been found.

By default, 4D defines a HTML Root folder named **WebFolder**. If it does not already exist, the HTML root folder is physically created on disk at the moment the Web server is launched for the first time.
If you keep the default location, the root folder is created:

- with 4D in local mode and 4D Server, at the same level as that of the database structure file.
  **Note:** as part of a compiled and merged application, the structure file is placed in the **Database** subfolder.
- with 4D in remote mode, in the local folder of the 4D database (see the **Get 4D folder** command).

You can modify the default HTML root folder name and location in the Database Settings dialog box (**Web** theme, **Configuration** page):




In the "Default HTML Root" entry area, enter the new access path of the folder that you wish to define.

The access path entered in this dialog box is relative: it is established from the folder containing the structure of the database (4D in local mode or 4D Server) or the folder containing the 4D application or software package (4D in remote mode).

For multi-platform compatibility of your databases, the 4D Web server uses particular writing conventions to describe access paths. The syntax rules are as follows:

- Folders are separated by a slash ("/")
- The access path must not end with a slash ("/")
- To "go up" one level in the folder hierarchy, enter ".." (two periods) before the folder name
- The access path must not start with a slash ("/") (except if you want the HTML root folder to be the database or 4D remote folder, see below).

For example, if you want the HTML root folder to be the "Web" subfolder in the "4DDatabase" folder, enter **4DDatabase/Web**.

If you want the HTML root folder to be the database or 4D remote folder, but for access to the folders above to be forbidden, enter "/" in the area. For a completely free access to the volumes, leave the "Default HTML Root" area empty.

**WARNING:** If you do not define a default HTML Root folder in the Preferences dialog box, the folder that contains the structure file of the database or the 4D application will be used. **Be careful because in this case there are no access restrictions** (users can access all the volumes).

**Notes:**

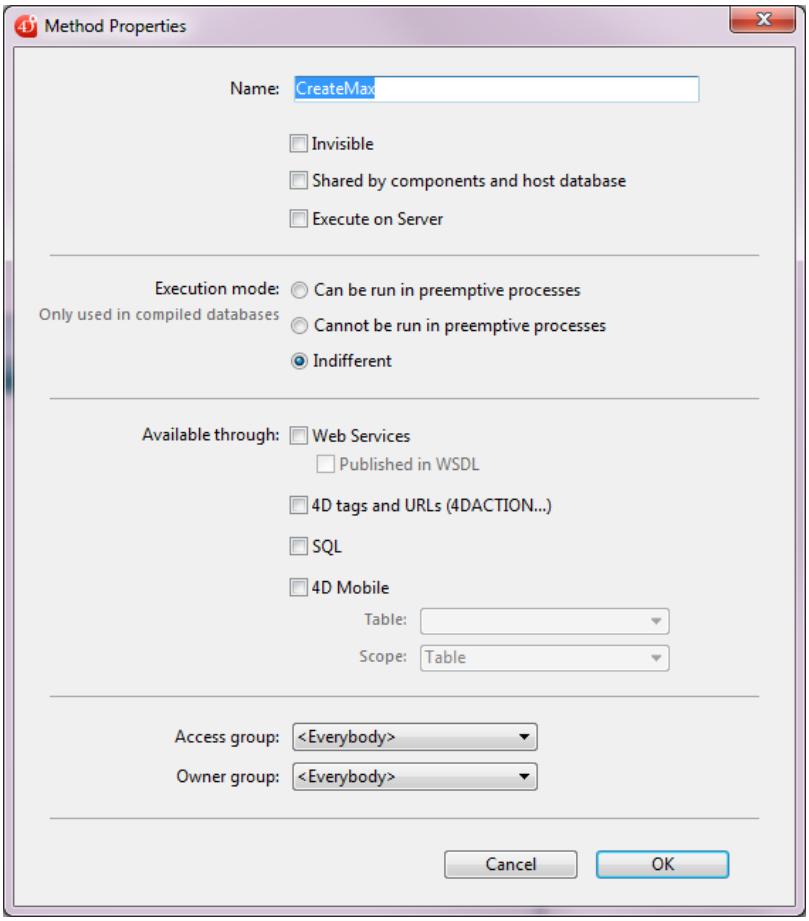
- When the HTML root folder is modified in the Database Settings dialog box, the cache is cleared so as to not store files whose access is restricted.
- You can also dynamically define the HTML root folder by using the **WEB SET ROOT FOLDER** command. In this case, the modification applies to all the current Web process for the worksession. The cache of the HTML pages is therefore cleared.

## Available through 4D tags and URLs

The special *4DACTION* URL and the *4DSCRIPT*, *4DEVAL, 4DTEXT, 4DHTML* (as well as the former *4DVAR* and *4DHTMLVAR*) tags, allow you to trigger the execution of any project method of a 4D database published on the Web. For example, the request *http://www.server.com/4DACTION/Erase_All* causes the execution of the ***Erase_All*** project method, if it exists.

This mechanism therefore presents a security risk for the database, in particular if an Internet user intentionally (or unintentionally) triggers a method not intended for execution via the Web. You can avoid this risk in three ways:

- Restrict access to project methods using the 4D password system. Drawbacks: This system requires the use of 4D passwords and forbids any type of method execution (including using HTML tags).
- Filter the methods called via the URLS using the **On Web Authentication Database Method**. Drawbacks: If the database includes a great number of methods, this system may by difficult to manage.
- Use the **Available through 4D tags and URLs (4DACTION...)** option found in the Method properties dialog box:




This option is used to individually designate each project method that can be called using the special URL, *4DACTION*, or the *4DSCRIPT*, *4DEVAL, 4DTEXT and 4DHTML* (as well as *4DVAR* and *4DHTMLVAR*) tags. When it is not checked, the project method concerned cannot be executed using an HTTP request containing a special URL or tag. Conversely, it can be executed using other types of calls (formulas, other methods, etc.).

This option is unchecked by default for databases created. Methods that can be executed using the *4DACTION* Web URL or the tags must be specifically indicated.

In the Explorer, Project methods with this property are given a specific icon:

## Allow database access through 4DSYNC URLs

This option on the "Web/Configuration" page of the Database SEttings lets you control support of requests containing /4DSYNC URLs. These URLs are used for synchronizing data through HTTP ((for more information about this mechanism, refer to **URL 4DSYNC/**).

This option enables or disables specific processing of requests containing /4DSYNC:

- When it is not checked, /4DSYNC requests are considered as standard requests and do not allow specific processing (using a synchronization request causes a "404 - resource unavailable" type response to be sent).
- When it is checked, the synchronization mechanism is enabled; /4DSYNC requests are considered as special requests and are parsed by the 4D HTTP server.

By default:

- this option is **not checked** in databases created with 4D beginning with version 13.
- this option is **checked** in databases converted from a previous version of 4D, for compatibility reasons. We recommend that you deselect it if your application does not use the HTTP replication function.

The scope of this option is local to the application and the Web server must be restarted to take it into account.

# On Web Authentication Database Method

## Description

The **On Web Authentication Database Method** is in charge of managing Web server engine access. It is called by 4D or 4D Server when a Web browser request requires the execution of a 4D method on the server (method called using a *4DACTION* URL, a *4DSCRIPT* tag, etc.).

This method receives six Text parameters: $1, $2, $3, $4, $5, and $6, and returns one Boolean parameter, $0. The description of these parameters is as follows:

| Parameters | Type | Description |
|---|---|---|
| $1 | Text | URL |
| $2 | Text | HTTP header + HTTP body (32 KB maximum) |
| $3 | Text | IP address of the Web client (browser) |
| $4 | Text | IP address of the server |
| $5 | Text | User name |
| $6 | Text | Password |
| $0 | Boolean | True = request accepted, False = request rejected |

You must declare these parameters as follows:

```
` On Web Authentication Database Method

C_TEXT($1;$2;$3;$4;$5;$6)
C_BOOLEAN($0)

` Code for the method
```

**Note:** All the On Web Authentication database method's parameters are not necessarily filled in. The information received by the database method depends on the options that you have previously selected in the Database Settings dialog box (please refer to the section **Connection Security**).

- **URL**

  The first parameter (*$1*) is the URL entered by the user in the location area of his or her Web browser, from which the host address has been removed.

  Let's take the example of an Intranet connection. Suppose that the IP address of your 4D Web Server machine is *123.4.567.89*. The following table shows the values of *$1* depending on the URL entered in the Web browser:

| URL entered in Web browser Location area | Value of parameter $1 |
|---|---|
| 123.4.567.89 | / |
| http://123.4.567.89 | / |
| 123.4.567.89/Customers | /Customers |
| http://123.4.567.89/Customers | /Customers |
| http://123.4.567.89/Customers/Add | /Customers/Add |
| 123.4.567.89/Do_This/If_OK/Do_That | /Do_This/If_OK/Do_That |

- **Header and Body of the HTTP request**

  The second parameter (*$2*) is the header and the body of the HTTP request sent by the Web browser. Note that this information is passed to your **On Web Authentication Database Method** as it is. Its contents will vary depending on the nature of the Web browser which is attempting the connection.
  If your application deals with this information, it is up to you to parse the header and the body.

**Notes:**

- For performance reasons, the size of data passing through the $2 parameter must not exceed 32 KB. Beyond this size, they are truncated by the 4D HTTP server.
- For more information about this parameter, please refer to the description of the **On Web Connection database method**.

- **Web client IP address**

  The $3 parameter receives the IP address of the browser's machine. This information can allow you to distinguish between Intranet and Internet connections.
  **Note:** 4D returns IPv4 addresses in a hybrid IPv6/IPv4 format written with a 96-bit prefix, for example ::ffff:192.168.2.34 for the IPv4 address 192.168.2.34. For more information, refer to the **Support of IPv6** section.

- **Server IP address**

  The $4 parameter receives the IP address used to call the Web server. 4D since version 6.5 allows for multi-homing, which allows you to exploit machines with more than one IP address. For more information, please refer to the section **Web Server Settings**

- **User Name and Password**

  The $5 and $6 parameters receive the user name and password entered by the user in the standard identification dialog box displayed by the browser. This dialog box appears for each connection, if a password management option has been selected in the Database Settings dialog box (see section **Connection Security**).

**Note:** If the user name sent by the browser exists in 4D, the $6 parameter (the user's password) is not returned for security reasons.

- **$0 parameter**

  The **On Web Authentication Database Method** returns a boolean in $0:
  - If $0 is **True**, the connection is accepted.
  - If $0 is **False**, the connection is refused.

The **On Web Connection database method** is only executed if the connection has been accepted by **On Web Authentication**.

**WARNING:** If no value is set to *$0* or if *$0* is not defined in the **On Web Authentication Database Method**, the connection is considered as accepted and the **On Web Connection database method** is executed.

**Notes :**

- Do not call any interface elements in the **On Web Authentication Database Method** (**ALERT**, **DIALOG**, etc.) because otherwise its execution will be interrupted and the connection refused. The same thing will happen if an error occurs during its processing.
- It is possible to prevent execution by *4DACTION* or *4DSCRIPT* for each project method via the "Available through 4D HTML tags and URLs (4DACTION...)" option in the Method properties dialog box. For more information about this point, please refer to the **Connection Security** section.

## On Web Authentication Database Method calls

The **On Web Authentication Database Method** is automatically called, regardless of the mode, when a request or processing requires the execution of a 4D method. It is also called when the Web server receives an invalid static URL (for example, if the static page requested does not exist).

The **On Web Authentication Database Method** is therefore called in the following cases:

- when 4D receives a URL beginning with *4DACTION/*
- when 4D receives a URL beginning with *4DCGI/*
- when 4D receives a URL beginning with *4DSYNC/*
- when 4D receives a URL requesting a static page that does not exist
- when 4D receives a root access URL and no home page has been set in the Database Settings or by means of the **WEB SET HOME PAGE** command
- when 4D processes a *4DSCRIPT* tag in a semi-dynamic page
- when 4D processes a *4DLOOP* tag based on a method in a semi-dynamic page.

**Compatibility note:** The database method is also called when 4D receives a URL beginning with *4DMETHOD/*. This URL is obsolete and is only kept for compatibility's sake.

Note that the **On Web Authentication Database Method** is NOT called when the server receives a URL requesting a valid static page.

## Example 1

Example of the **On Web Authentication Database Method** in BASIC mode:

```
 `On Web Authentication Database Method
C_TEXT($5;$6;$3;$4)
C_TEXT($user;$password;$BrowserIP;$ServerIP)
C_BOOLEAN($4Duser)
ARRAY TEXT($users;0)
ARRAY LONGINT($nums;0)
C_LONGINT($upos)
C_BOOLEAN($0)

$0:=False

$user:=$5
$password:=$6
$BrowserIP:=$3
$ServerIP:=$4

 `For security reasons, refuse names that contain @
If(WithWildcard($user)|WithWildcard($password))
   $0:=False
 `The WithWildcard method is described below
Else
 `Check to see if it' s a 4D user
   GET USER LIST($users;$nums)
   $upos:=Find in array($users;$user)
   If($upos >0)
      $4Duser:=Not(Is user deleted($nums{$upos}))
   Else
      $4Duser:=False
   End if

   If(Not($4Duser))
 `It is not a user defined 4D, look in the table of Web users
      QUERY([WebUsers];[WebUsers]User=$user;*)
      QUERY([WebUsers]; & [WebUsers]Password=$password)
      $0:=(Records in selection([WebUsers])=1)
   Else
      $0:=True
   End if
End if
 `Is this an intranet connection?
If(Substring($BrowserIP;1;7)#"192.100.")
   $0:=False
End if
```

## Example 2

Example of the **On Web Authentication Database Method** in DIGEST mode:

```
 // On Web Authentication Database Method
C_TEXT($1;$2;$5;$6;$3;$4)
C_TEXT($user)
C_BOOLEAN($0)
$0:=False
$user:=$5
 // For security reasons, refuse names that contain @
If(WithWildcard($user))
   $0:=False
 // The <span class="rte4d_met">WithWildcard</span> method is described below
Else
   QUERY([WebUsers];[WebUsers]User=$user)
   If(OK=1)
      $0:=WEB Validate digest($user;[WebUsers]password)
   Else
      $0:=False // User does not exist
   End if
End if
```

The WithWildcard method is as follows:

```
// WithWildcard Method
// WithWildcard ( String ) -> Boolean
// WithWildcard ( Name ) -> Contains a Wilcard character

C_LONGINT($i)
C_BOOLEAN($0)
C_TEXT($1)

$0:=False
For($i;1;Length($1))
   If(Character code(Substring($1;$i;1))=Character code("@"))
      $0:=True
   End if
End for
```

## On Web Connection Database Method

---

The **On Web Connection Database Method** can be called in the following cases:

- the Web server receives a request beginning with the *4DCGI* URL.
- the Web server receives an invalid request.

For more information, refer to the paragraph “On Web Connection Database Method calls” below.

**Compatibility note**: The database method is also called when a context is created in contextual mode (obsolete mode that could be used in converted 4D databases).

The request should have been previously accepted by the **On Web Authentication database method** (if it exists) and the Web server must be launched.

The **On Web Connection Database Method** receives six text parameters that are passed by 4D. The contents of these parameters are as follows:

| Parameters | Type | Description |
|---|---|---|
| $1 | Text | URL |
| $2 | Text | HTTP header + HTTP body (up to 32 kb limit) |
| $3 | Text | IP address of the Web client (browser) |
| $4 | Text | IP address of the server |
| $5 | Text | User name |
| $6 | Text | Password |

You must declare these parameters as follows:

```
` On Web Connection Database Method

C_TEXT($1;$2;$3;$4;$5;$6)

` Code for the method
```

- **URL extra data**

  The first parameter (*$1*) is the URL entered by the user in the location area of his or her Web browser, from which the host address has been removed.
  Let’s take the example of an Intranet connection. Suppose that the IP address of your 4D Web Server machine is *123.4.567.89*. The following table shows the values of *$1* depending on the URL entered in the Web browser:

  | URL entered in Web browser Location area | Value of parameter $1 |
  |---|---|
  | 123.4.567.89 | / |
  | http://123.4.567.89 | / |
  | 123.4.567.89/Customers | /Customers |
  | http://123.4.567.89/Customers | /Customers |
  | http://123.4.567.89/Customers/Add | /Customers/Add |
  | 123.4.567.89/Do_This/If_OK/Do_That | /Do_This/If_OK/Do_That |

  Note that you are free to use this parameter at your convenience. 4D simply ignores the value passed beyond the host part of the URL.
  For example, you can establish a convention where the value *"/Customers/Add"* means “go directly to add a new record in the *[Customers]* table.” By supplying the Web users of your database with a list of possible values and/or default bookmarks, you can provide shortcuts to the different parts of your application. This way, Web users can quickly access resources of your Web site without going through the whole navigation path each time they make a new connection to your database.

**WARNING:** In order to prevent a user from reentering a database with a bookmark created during a previous session, 4D intercepts any URL that corresponds to one of the standard 4D URLs.

- **Header of the HTTP request followed by the HTTP body**
  The second parameter (*$2*) is the header and the body of the HTTP request sent by the Web browser. Note that this information is passed to your **On Web Connection Database Method** as it is. Its contents will vary depending on the nature of the Web browser which is attempting the connection.

  With Safari running on Mac OS, you may receive a header similar to this:

  ```
  GET /favicon.ico HTTP/1.1
  Referer: http://123.45.67.89/4dcgi/test
  User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X; fr-fr) AppleWebKit/523.10.3 (KHTML, like Gecko) Version/3.0.4
  Safari/523.10
  Cache-Control: max-age=0
  Accept: */*
  Accept-Language: fr-fr
  Accept-Encoding: gzip, deflate
  Connection: keep-alive
  Host: 123.45.67.89
  ```

  With Microsoft Internet Explorer 8 running on Windows, you may receive a header similar to this:

  ```
  GET / HTTP/1.1
  Accept: image/jpeg, application/x-ms-application, image/gif, application/xaml+xml, image/pjpeg, application/x-ms-xbap,
  application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, */*
  Accept-Language: fr-FR
  User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR
  3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C)
  Accept-Encoding: gzip, deflate
  Host: 123.45.67.89
  Connection: Keep-Alive
  ```

  If your application deals with this information, it is up to you to parse the header and the body.
  **Note:** For performance reasons, the size of these data cannot be more than 32 KB. Beyond this, they are truncated by the 4D HTTP server.

- **IP address of the Web client**
  The *$3* parameter receives the IP address of the browser's machine. This information can allow you to distinguish between Intranet and Internet connections.
  **Note:** 4D returns IPv4 addresses in a hybrid IPv6/IPv4 format written with a 96-bit prefix, for example ::ffff:192.168.2.34 for the IPv4 address 192.168.2.34. For more information, refer to the **Support of IPv6** section.

- **IP address of the server**
  The *$4* parameter receives the IP address to which the HTTP request was sent. 4D allows for multi-homing, which allows you to exploit machines with more than one IP address. For more information, please refer to the section **Web Server Settings**.

- **User Name and Password**
  The *$5* and *$6* parameters receive the user name and password entered by the user in the standard identification dialog box displayed by the browser. This dialog box appears for each connection, if the **Use Passwords** option has been selected in the Database Settings dialog box (see section **Connection Security**).

**Note:** If the user name sent by the browser exists in 4D, the *$6* parameter (the user's password) is not returned for security reasons.

## On Web Connection Database Method Calls

The **On Web Connection Database Method** can be used as the entry point for the 4D Web server, either using the special *4DCGI* URL, or using customized command URLs.

**Warning:** Calling a 4D command that displays an interface element (**DIALOG**, **ALERT**, etc.) ends the method processing.

The **On Web Connection Database Method** is therefore called in the following cases:

- When 4D receives the */4DCGI* URL. The database method is called with the */4DCGI/<action>* URL in *$1*.
- When a Web page is called with a URL of type *<path>/<file>* is not found. The database method is called with the URL (*).
- When a Web page is called with a URL of type *<file>/* and no home page has been defined by default. The database method is called with the URL (*).

(*) In this particular cases, the URL received in *$1* does NOT start with the "/" character.

## On Web Close Process database method

---

The **On Web Close Process database method** is called by the 4D Web server each time a Web session is about to be closed. A session can be closed in the following cases:

- when the maximum number of simultaneous sessions is reached (100 by default, modifiable using the **WEB SET OPTION** command), and 4D needs to create new ones (4D automatically kills the process of the oldest inactive session),
- when the maximum period of inactivity for the session process is reached (480 minutes by default, modifiable using the **WEB SET OPTION** command),
- when the **WEB CLOSE SESSION** command is called.

When this database method is called, the context of the session (variables and selections generated by the user) is still valid. This means that you can save data related to the session in order to be able to use them again subsequently, more specifically using the **On Web Connection Database Method**.

**Note:** In the context of a 4D Mobile session (which can generate several processes), the **On Web Close Process database method** is called for each Web process that is closed, allowing you to save all types of data (variables, selection, etc.) generated by the 4D Mobile session process.

An example of the **On Web Close Process database method** is provided in the **Web Sessions Management** section.

# Web Sessions Management

The 4D Web Server provides a simple and complete mechanism for managing user sessions. This automatic mechanism allows successive Web clients to reuse the same context (selections and variable instances) from one request to another.

This is done through a private cookie set by 4D itself: "4DSID". On each web client request, 4D checks for the presence and the value of the 4DSID cookie:

- If the cookie has a value, 4D tries to find the context that created this cookie among the existing contexts,
  - If this context is found, it is reused for the call; the ***Compiler_Web*** method is not executed,
  - If no context is found, 4D creates a new one,
- If the cookie does not have a value or if it is not present (because it is expired, for instance), 4D creates a new context.

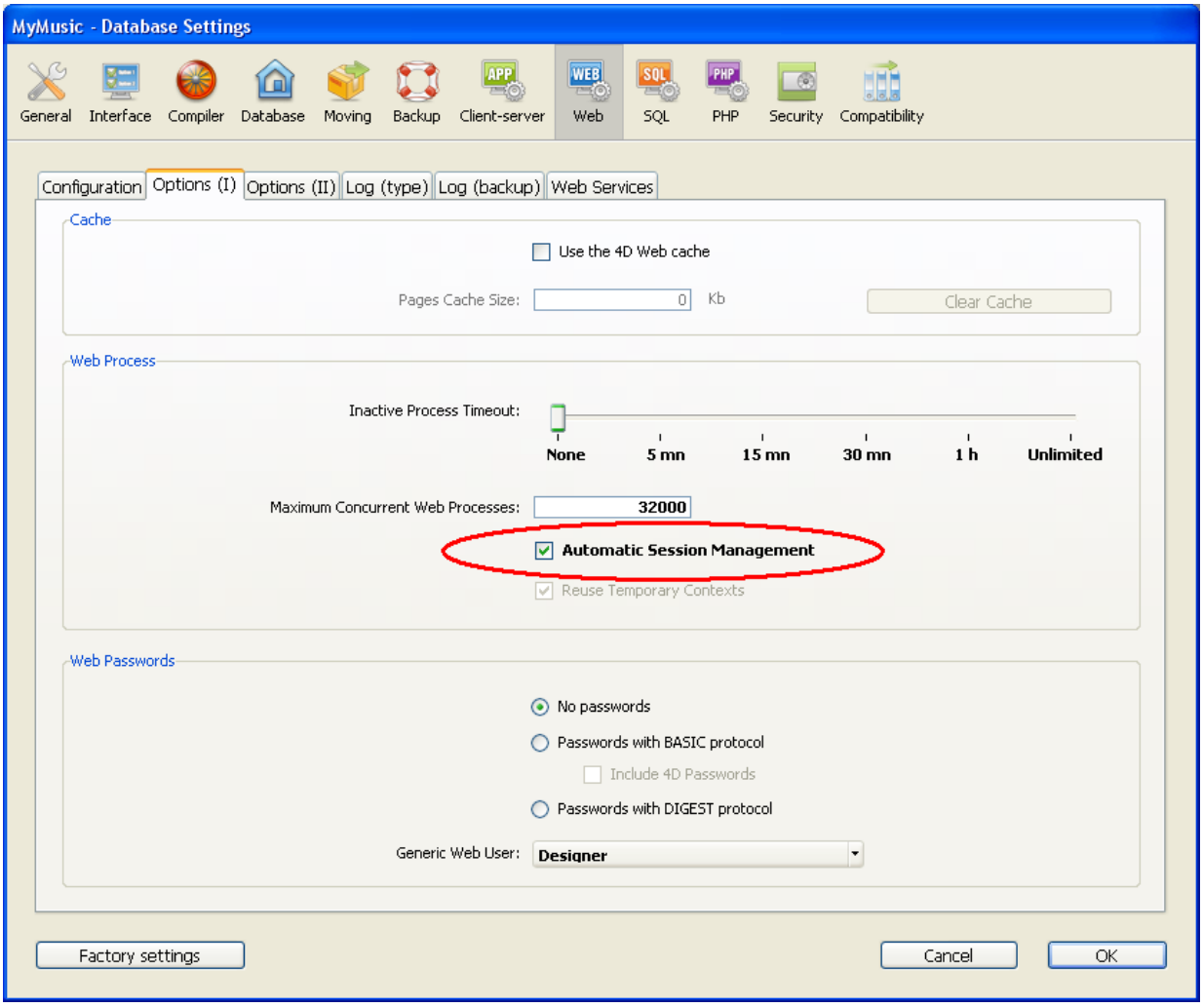
## Enabling and disabling the mechanism

The session management mechanism must be enabled on your 4D Web server so that you can use it in your application.

By default, this mechanism is enabled in databases created with 4D v13 and later versions. However, for compatibility reasons, it is disabled in databases converted from previous versions of 4D. You must enable it explicitly in order to benefit from this new functionality.

There are two ways to enable automatic session management:

- Using the **Automatic Session Management** option on the "Options (I)" page of the Database Settings:




  When you do this, the setting is permanent; it is saved to disk.
- Using the Web Keep session option of the **WEB SET OPTION** command. In this case, this setting is applied only to the session and "overrides" the setting defined in the database settings.

In both cases, the setting is local to the machine; so it can be different on the 4D Server Web server and the Web servers of remote 4D machines.

## Cookie expiration and preserving of contexts

The lifespan of an inactive cookie is 8 hours (480 minutes) by default but this can be changed using the **WEB SET OPTION** command. You can set a different lifespan for cookies (Web Inactive session timeout option) from the one for processes associated with server sessions (Web Inactive process timeout option): for instance, you may want for a shopping "basket" to remain valid for 24 hours but, for optimization purposes, you do not want to keep the process for that long. In this case, you can set a process life duration of 4 hours, for example. At the end of this period, the **On Web Close Process database method** is called and you can store the variables and selections related to the session before the process is killed. The next time the Web client connects (up to 24 hours later), the cookie is sent back to the server and you can reload the session information in the **On Web Connection Database Method** (see example below).

If necessary, you can use the **WEB CLOSE SESSION** command to force the expiration of the cookie at any time and thus close the session.

## Destruction of inactive contexts

4D automatically destroys the oldest inactive contexts when the maximum number of kept contexts is reached (this number is 100 by default and can be changed using the **WEB SET OPTION** command).
When a context is about to be destroyed (closing of the associated Web process), the **On Web Close Process database method** is called, which lets you save the context variables and selections, in anticipation of subsequent reuse.

## Example

This example shows how easy it is to manage sessions using the **On Web Connection Database Method**and the **On Web Close Process database method**.

Here is the code of the **On Web Connection Database Method**:

```
 // On Web Connection (or On Web Authentication)
C_TEXT(www_SessionID)
If(www_SessionID=WEB Get Current Session ID)
 // Compiler_Web is not called
 // All variables and selection already exist
   ...
Else
 // Compiler_Web has just been executed.
 // This is a new session, no variable or selection exists
 // Keep track of the session that 4D just created
   www_SessionID:=WEB Get Current Session ID

 // Initialization of session
 // Set up selections
 // find connected user
   QUERY([User];[User]Login=www_Login)
   QUERY([prefs];[prefs]Login=www_Login)

 // find employee coordinates
   QUERY([employees];[employees]Name=[user]name)
   QUERY([company];[company]Name=[user]company)

 // Setup variables
 // Get prefs for this user
   SELECTION TO ARRAY([prefs]name;prefNames;[prefs]values;prefValues)
   www_UserName:=[User]Name
   www_UserMail:=[User]mail

 // Session is now initialized
End if
```

Code for **On Web Close Process database method** :

```
 // On Web Session Suspend
 // After a period of inactivity or whenever necessary, 4D closes the session
```

```
C_TEXT(www_SessionID)
www_SessionID:=""
 // We store the session info
 // We save the preferences of the previously connected user
QUERY([prefs];[prefs]Login=www_Login) // kept in the session
ARRAY TO SELECTION(prefNames;[prefs]name;prefValues;[prefs]values)

 // Important: the process is then killed
 // 4D deletes the variables, selections, etc.
```

## In Case of Rejected Cookies

Since the session management mechanism is based on the use of cookies, the 4D HTTP server is not able to maintain a session if the Web client does not accept cookies. In this case, each request is treated as a new connection and the ***Compiler_Web*** method is executed on each connection.

You can check for cookie support with the **WEB GET HTTP HEADER** command.

## Session and IP management

The 4D HTTP Server keeps track of the IP that started a session. If a different IP tries to access an existing session, an HTTP 400 error is returned to the client.

# Semi-dynamic pages

4D's Web server allows you to use **semi-dynamic pages**.

These pages are HTML 'templates' containing **4D Transformation Tags**, i.e. a mix of static HTML code and 4D references added by means of transformation tags such as 4DHTML, 4DIF, or 4DINCLUDE. These tags are inserted as HTML type comments (*<!--#Tag Contents-->*) into the HTML source code.

**Note:** An alternative $-based syntax is used in certain conditions for 4DHTML, 4DTEXT and 4DEVAL tags in order to make them XML-compliant. For more information, refer to **Alternative syntax for 4DTEXT, 4DHTML, 4DEVAL** section.

When these pages are sent by the HTTP server, they are parsed and the tags they contain are executed and replaced with the resulting data. The pages received by the browsers are thus a combination of static elements and values coming from 4D.

## Principles

You can use programming to give default values to HTML objects by including *<!--#4DTEXT VarName-->* in the **value** field of the HTML object, where **VarName** is the name of the 4D process variable as defined in the current Web process. This is the name that you surround with the standard HTML notation for comments *<!--#...-->*.

**Note**: Some HTML editors may not accept *<!--#4DTEXT VarName-->* in the **value** field of HTML objects. In this case, you will have to type it in the HTML code.

In fact, the syntax *<!--#4DTEXT VarName-->* allows you to insert 4D data anywhere in the HTML page. For example, if you write:

<P>Welcome to *<!--#4DTEXT vtSiteName-->*!</P>

The value of the 4D variable *vtSiteName* will be inserted in the HTML page.

Here is an example:

```
 // The following piece of 4D code assigns "4D4D" to the process variable vs4D
vs4D:="4D4D"
 // Then it sends the HTML page "AnyPage.HTM"
SEND HTML FILE("AnyPage.HTM")
```

The source of the HTML page **AnyPage.HTM** is listed here:

```
<html> <head>     <title>AnyPage</title> <script language="JavaScript"><!-- function Is4DWebServer(){     return (document.frm.vs4D.value=="4D4D") } function HandleButton(){     if(Is4DWebServer()){         alert("You are connected to 4D Web Server!")     } else {         alert("You are NOT connected to 4D Web Server!") } //--></script> </head> <body> <form action="/4DACTION/WWW_STD_FORM_POST" method="post" name="frm"> <p><input type="hidden" name="vs4D" value="<!--#4DTEXT vs4D-->"</p> <p><a href="JavaScript:HandleButton()"><img src="AnyGIF.GIF" border=0 align=bottom></a></p> <p><input type="submit" name="bOK" value="OK"></p> </form> </body> </html>
```

The *<!--#4DTEXT -->* tag also allows the insertion of **4D expressions** in the pages sent (fields, array elements, etc.). The operation of this tag with this type of data is identical to that with variables. You can also insert HTML code into 4D variables using the *4DHTML* tag. Other tags such as *4DIF* allow you to control the executed code. A description of all the tags that you can use is found in the **4D Transformation Tags** section.

## Processing tags

Parsing of the contents of semi-dynamic pages sent by 4D takes place when **WEB SEND FILE** (.htm, .html, .shtm, .shtml), **WEB SEND BLOB** (text/html type BLOB) or **WEB SEND TEXT** commands are called, as well as when sending pages called using URLs. In this last case, for reasons of optimization, pages that are suffixed with “.htm” and “.html” are NOT parsed. In order to "force" the parsing of HTML pages in this case, you must add the suffix “.shtm” or “.shtml” (for example, http://www.server.com/dir/page.shtm). An example of the use of this type of page is given in the description of the **WEB GET STATISTICS** command. XML pages (.xml, .xsl) and WML pages (.wml) are also supported and always parsed by 4D (see **XML and WML Support** section).

You can also carry out parsing outside of the Web context when you use the **PROCESS 4D TAGS** command.

Internally, the parser works with UTF-16 strings, but the data to parse may have been encoded differently. When tags contain text  (for example, 4DHTML), 4D converts the data when necessary depending on its origin and the information

available (*charset*). Below are the cases where 4D parses the tags contained in the HTML pages, as well as any conversions carried out:

| Action | Content analysis of the sent pages | Support of $ syntax(*) | Character set used for parsing tags |
|---|---|---|---|
| Pages called via URLs | X, except for pages with ".htm" or ".html" extensions | X, except for pages with ".htm" or ".html" extensions | Use of charset passed as parameter of the "Content-Type" header of the page. If there is none, search for a META-HTTP EQUIV tag with a charset. Otherwise, use of default character set for the HTTP server |
| **WEB SEND FILE** command call | X | - | Use of charset passed as parameter of the "Content-Type" header of the page. If there is none, search for a META-HTTP EQUIV tag with a charset. Otherwise, use of default character set for the HTTP server |
| **WEB SEND TEXT** command call | X | - | No conversion necessary |
| **WEB SEND BLOB** command call | X, if BLOB is of the "text/html" type | - | Use of charset set in the "Content-Type" header of the response. Otherwise, use of default character set for the HTTP server |
| Inclusion by the *<!--#4DINCLUDE-->* tag | X | X | Use of charset passed as parameter of the "Content-Type" header of the page. If there is none, search for a META-HTTP EQUIV tag with a charset. Otherwise, use of default character set for the HTTP server |
| **PROCESS 4D TAGS** command call | X | X | Text data: no conversion. BLOB data: automatic conversion from the Mac-Roman character set for compatibility |

(*) The alternative $-based syntax is available for 4DHTML, 4DTEXT and 4DEVAL tags (see **Alternative syntax for 4DTEXT, 4DHTML, 4DEVAL** section).

## JavaScript Encapsulation

4D supports JavaScript source code embedded into HTML documents, and also JavaScript *.js* files embedded in HTML documents (for example *<SCRIPT SRC="...">*).

Using **WEB SEND FILE** or **WEB SEND BLOB**, you send a page that you have prepared in an HTML source editor or built programmatically using 4D and saved as a document on disk. In both cases, you have full control of the page. You can insert JavaScript scripts in the HEAD section of the document as well as use scripts with the FORM markup. In the previous example, the script refers to the form "frm" because you were able to name the form. You can also trigger, accept, or reject the submission of the form at the FORM markup level.

**Note**: 4D supports Java applets transport.

# URLs and Form Actions

The 4D Web Server offers different URLs and HTML form actions that allow you to implement various actions in your database.

These URLs are the following:

- *4DACTION/*, to link any HTML object to a project method of your database,
- *4DCGI/*, to call the **On Web Connection Database Method** from any HTML object.
- *4DSYNC/*, to synchronize the data of the tables.

In addition, the 4D Web Server accepts several additional URLs:

- */4DSTATS*, */4DHTMLSTATS*, */4DCACHECLEAR* and */4DWEBTEST*, to allow you to obtain information about the functioning of your 4D Web site. These URLs are described in the section **Information about the Web Site**.
- */4DWSDL*, to allow you to access the declaration file of Web Services published on the server. For more information, refer to the **Web Services (Server) Commands** section and to the Design Reference manual.

## URL 4DACTION/

**Syntax:** *4DACTION/MyMethod{/Param}*

**Usage:** URL or Form action.

This URL allows you to link an HTML object (text, button...) to a 4D project method. The link will be */4DACTION/MyMethod/Param* where ***MyMethod*** is the name of the 4D project method to be executed when the user clicks on the link and *Param* an optional Text parameter to pass to the method in *$1* (see paragraph “The Text Parameters Passed to 4D Methods Called via URLs” below).

When 4D receives a */4DACTION/MyMethod/Param* request, the **On Web Authentication Database Method** (if it exists) is called. If it returns **True**, the ***MyMethod*** method is executed.
*4DACTION/* can be associated with a URL in a static Web page. The syntax of the URL must be in the following form:

```
<A HREF="/4DACTION/MyMethod/Param"> Do Something</A>
```

The ***MyMethod*** project method should generally return a "reply" (sending of an HTML page using **WEB SEND FILE** or **WEB SEND BLOB**, etc.). Be sure to make the processing as short as possible in order not to block the browser.

**Note:** A method called by *4DACTION* must not call interface elements (**DIALOG**, **ALERT**, etc.).

**Warning:** For a 4D method to be able to be executed using the *4DACTION/* URL, it must have the “Available through 4D HTML tags and URLS (4DACTION...)” attribute (unchecked by default), defined in the Method properties. For more information on this point, refer to the section **Connection Security**.

### Example 1

This example describes the association of the *4DACTION/* URL with an HTML picture object in order to dynamically display a picture in the page. You insert the following instructions in a static HTML page:

```
<IMG SRC="/4DACTION/PICTFROMLIB/1000">
```

The ***PICTFROMLIB*** method is as follows:

```
C_TEXT($1) // This parameter must always be declared
C_PICTURE($PictVar)
C_BLOB($BlobVar)
C_LONGINT($Number)
 // We retrieve the picture' s number in the string $1
$Number:=Num(Substring($1;2;99))
GET PICTURE FROM LIBRARY($Number;$PictVar)
```

```
PICTURE TO GIF($PictVar;$BlobVar)
WEB SEND BLOB($BlobVar;"Pict/gif")
```

### 4DACTION to post forms

The 4D Web server also allows you to use “posted” forms, which are static HTML pages that send data to the Web server, and to easily retrieve all the values. The POST type must be associated to them and the form’s action must imperatively start with */4DACTION/MethodName*.

**Note:** A form can be submitted through two methods (both can be used with 4D):

- POST, usually used to add data into the Web server - to a database,
- GET, usually used to request the Web server - data coming from a database.

In this case, when the Web server receives a posted form, it calls the **On Web Authentication Database Method** (if it exists). If it returns **True**, the ***MethodName*** method is executed. In this method, you must call the **WEB GET VARIABLES** command in order to retrieve the names and values of all the fields included in an HTML page submitted to the server.

**Compatibility note:** In converted databases, if the "Automatic variable assignment" option on the **Compatibility page** is checked, the special ***COMPILER_WEB*** project method (if it exists) is called first; 4D retrieves the values of HTML fields found in the form and automatically fills the 4D variables in the called method with their contents if they have the same name. This functioning is obsolete. For more information, refer to the **Binding 4D objects with HTML objects** section.

The HTML syntax to apply in the form is of the following type:

- to define the action in a form:

```
<FORM ACTION="/4DACTION/MethodName" METHOD=POST>
```

- to define a field in a form:

```
<INPUT TYPE=Field type NAME=Field name VALUE="Default value">
```

For each field in the form, 4D sets the value of the field to the variable with the same name.

### Example 2

In a 4D Web database, we would like for the browsers to be able to search among the records by using a static HTML page. This page is called “search.htm”. The database contains other static pages that allow you to, for example, display the search result (“results.htm”). The POST type has been associated to the page, as well as the */4DACTION/SEARCH* action.

Here is the HTML code that corresponds to this page:

```
<FORM ACTION="/4DACTION/PROCESSFORM" METHOD=POST> <INPUT TYPE=TEXT NAME=VNAME VALUE=""><BR> <INPUT TYPE=CHECKBOX NAME=EXACT
VALUE="Word">Whole word<BR> <INPUT TYPE=SUBMIT NAME=OK VALUE="Search"> </FORM>
```

During data entry, type “ABCD” in the data entry area, check the "Whole word" option and validate it by clicking the **Search** button.

In the request sent to the Web server:

```
VNAME="ABCD"
vEXACT="Word"
OK="Search"
```

4D calls the **On Web Authentication Database Method** (if it exists), then the PROCESSFORM project method is called, which is as follows:

```
C_TEXT($1) //mandatory for compiled mode
C_LONGINT($vName)
C_TEXT(vNAME;vLIST)
ARRAY TEXT($arrNames;0)
ARRAY TEXT($arrVals;0)
WEB GET VARIABLES($arrNames;$arrVals) //we retrieve all the variables of the form
$vName:=Find in array($arrNames;"vNAME")
vNAME:=$arrVals{$vName}
If(Find in array($arrNames;"vEXACT")=-1) //If the option has not been checked
```

```
      vNAME:=VNAME+"@"
   End if
   QUERY([Jockeys];[Jockeys]Name=vNAME)
   FIRST RECORD([Jockeys])
   While(Not(End selection([Jockeys])))
      vLIST:=vLIST+[Jockeys]Name+" "+[Jockeys]Tel+"<BR>"
      NEXT RECORD([Jockeys])
   End while
   WEB SEND FILE("results.htm") //Send the list to the results.htm form
    //which contains a reference to the variable vLIST,
    //for example <!--4DHTML vLIST-->
    //...
  End if
```

## URL 4DCGI/

**Syntax:** *4DCGI/<action>*

**Usage:** URL.

When the 4D Web server receives the */4DCGI/<action>* URL, the **On Web Authentication Database Method**(if it exists) is called. If it returns **True**, the Web server calls the **On Web Connection Database Method** by sending the URL "as is " to *$1*.

The *4DCGI/* URL does not correspond to any file. Its role is to call 4D using the **On Web Connection Database Method**. The "<action>" parameter can contain any type of information.

This URL allows you to perform any type of action. You just need to test the value of *$1* in the **On Web Connection Database Method** or in one of its submethods and have 4D perform the appropriate action. For example, you can build completely custom static HTML pages to add, search, or sort records or to generate GIF images on-the-fly. Examples of how to use this URL are in the descriptions of the **PICTURE TO GIF** and **WEB SEND HTTP REDIRECT** commands.

When issuing an action, a "response" must be returned, by using commands that send data (**WEB SEND FILE**, **WEB SEND BLOB**, etc.).

**Warning:** Please be sure to execute the shortest possible actions so as not to hold up the browser.

## The Text Parameters Passed to 4D Methods Called via URLs

4D sends text parameters to any 4D method called via special URLs (*4DACTION/* and *4DCGI/*), in both contextual and non-contextual modes. Regarding these text parameters:

- Although you do not use these parameters, you must explicitly declare them with the **C_TEXT** command, otherwise runtime errors will occur while using the Web to access a database that runs in compiled mode. The message is of the following type:
  *"**Error in dynamic code***
  *Invalid parameters in an EXECUTE command*
  *Method Name:*
  *Line Number:*
  *Description: [<date and time>]"*
  This runtime error is caused by the fact that the text parameter *$1* was not declared in the 4D method called when you clicked on the HTML link. Since the execution context is the current HTML page, the error does not specifically reference the method line. Explicitly declaring the text parameter *$1* will eliminate these errors:

  ```
   //M_SEND_PAGE project method
  C_TEXT($1) // This parameter MUST be explicitly declared
   //...
  WEB SEND FILE($mypage)
  ```

- The *$1* parameter returns the extra data placed at the end of the URL, and can be used as a placeholder for passing values from the HTML environment to the 4D environment.

### Parameters to declare explicitly in the called 4D method

You must declare different parameters depending on the nature and the origin of the call to a 4D method.

- **On Web Authentication Database Method** (if any) and **On Web Connection Database Method**
  You must declare the six parameters of the connection:

```
 //On Web Connection Database Method
C_TEXT($1;$2;$3;$4;$5;$6) //These parameters MUST be explicitly declared
```

- Method called by the URL *4DACTION/*
  You must declare the *$1* parameter:

```
 //Method called by the URL 4DACTION/
C_TEXT($1) //This parameter MUST be explicitly declared
```

- Method called by the tag *4DSCRIPT/* as an HTML comment in a document
  The method should return a value in $0. You must declare the *$0* and *$1* parameter:

```
 //Method called by the tag 4DSCRIPT/ as an HTML comment
C_TEXT($0;$1) //These parameters MUST be explicitly declared
```

## URL 4DSYNC/

**Syntax:**
*4DSYNC/$catalog{/TableName}*
*4DSYNC/TableName{/TableName}{/FieldName1,...,FieldNameN}{Params}*

**Usage:** URL in POST or GET method

This URL synchronizes the data in the tables of the local 4D database with a remote database using HTTP. It is used to synchronize a 4D database with a client application installed for example on a Smartphone or other third-party HTTP applications.

The *4DSYNC/* URL is used in a GET method to recover the data of the 4D database or in a POST method to update the data in the 4D database.

Here are the different HTTP requests that can be received:

- *GET /4DSYNC/$catalog*
  Returns the list of database tables and how many records they contain. For example, if you have a structure with two tables: PEOPLE (2 records) and INVOICES (3 records), when you use the syntax: *http://localhost/4DSYNC/$catalog/* it returns PEOPLE 2 INVOICES 3 in the browser.

- *GET /4DSYNC/$catalog/TableName*
  Returns a description of the *TableName* structure (XML format).

- *GET /4DSYNC/TableName/FieldName1{,FieldName2},...*
  Returns the data of the *FieldName* field in the *TableName* table.

- *GET /4DSYNC/TableName/FieldName1?$stamp=0&$format=json*
  Returns the data of the *FieldName* field in the *TableName* table starting from *stamp* 0 and in the *json* format.

- *POST /4DSYNC/TableName/FieldName1{,FieldName2},...*
  Integrates the modifications made on the client machine into the 4D database.

**Note:** The data exchange format is JavaScript Objet Notation (JSON). The complete grammar is available from the technical support of 4D, Inc.

**Note:** In order for synchronization mechanisms to be enabled, the **Allow database access through "4DSYNC" URLs** option must be checked on the "Web/Configuration" page of the Database Settings (see below). Otherwise, requests containing 4DSYNC URLs will fail.

### Notes about synchronization with HTTP

When using the *4DSYNC/* URL, you need to take the following principles into account:

- 4D works with the virtual structure of the database if it exists. In this case, the *TableName* and *FieldName* parameters correspond to table and field names that have been specified using the **SET FIELD TITLES** and **SET TABLE TITLES** commands. Note that the scope of these commands is the session and when using 4D Server you must call them in a stored procedure on the server.
- Replication of BLOB and Picture type fields is not supported.

- To be able to synchronize data:
  - The HTTP server must be launched.
  - The **Allow database access through "4DSYNC" URLs** option must be checked on the "Web/Configuration" page of the Database Settings (see the **Connection Security** section).
  - The "Enable Replication" property must be checked for each table where you want to synhronize the data. If a virtual structure has been defined, these tables must be included in it. Warning: checking this option publishes additional information; you must make sure that access to your database is protected (see the description of this option in **Connection Security**).
- When 4D receives a */4DSYNC* request, the **On Web Authentication Database Method** is called (except when the password is incorrect, see the connection diagram in **Connection Security**). If it returns **True**, the request is executed; otherwise it is refused.

# Binding 4D objects with HTML objects

4D's Web server lets you recover "posted" data, i.e. data entered by users through Web forms and sent to the server by means of buttons or interface elements, in POST mode or in GET mode.

The Web server accepts several specific URLs that can be associated with buttons so that the submission of the form triggers server-side processing. These URLs are described in the **URLs and Form Actions** section.

This chapter covers the principles implemented in order to recover data found in forms that were returned to the server.

## Receiving dynamic values

When the 4D Web server receives a posted form, 4D can retrieve the values of any HTML objects it contains. This principle can be implemented in the case of a Web form, sent for example using **WEB SEND FILE** or **WEB SEND BLOB**, where the user enters or modifies values, then clicks on the validation button. In this case, there are two ways that 4D can retrieve the values of the HTML objects found in the request:

- using the **WEB GET VARIABLES** command, or
- using the **WEB GET BODY PART** and **WEB Get body part count** commands.

The **WEB GET VARIABLES** command retrieves the values as text, while the **WEB GET BODY PART** and **WEB Get body part count** commands can retrieve the files posted, using BLOBs.

**Compatibility note (4D v13.4):** In previous versions, 4D copied the values of variables received by means of a posted Web form or a GET URL directly into 4D process variables (in compiled mode, these variables had to have been declared in the COMPILER_WEB method). This functioning is removed starting with 4D v13.4; it is maintained by compatibility in converted databases but can be disabled using the **Automatic variable assignment** compatibility option on the **Compatibility page** of the Database Settings (we recommend that you uncheck this option and use the **WEB GET VARIABLES** or **WEB GET BODY PART** commands in your databases).

Consider the following HTML page source code:

```
<html> <head>  <title>Welcome</title>  <script language="JavaScript"><!-- function GetBrowserInformation(formObj){ formObj.vtNav_appName.value = navigator.appName formObj.vtNav_appVersion.value = navigator.appVersion formObj.vtNav_appCodeName.value = navigator.appCodeName formObj.vtNav_userAgent.value = navigator.userAgent return true } function LogOn(formObj){ if(formObj.vtUserName.value!=""){ return true } else { alert("Enter your name, then try again.") return false } } //--></script> </head> <body> <form action="/4DACTION/WWW_STD_FORM_POST" method="post" name="frmWelcome" onsubmit="return GetBrowserInformation(frmWelcome)">  <h1>Welcome to Spiders United</h1>  <p><b>Please enter your name:</b> <input name="vtUserName" value="" size="30" type="text"></p>  <p> <input name="vsbLogOn" value="Log On" onclick="return LogOn(frmWelcome)" type="submit"> <input name="vsbRegister" value="Register" type="submit"> <input name="vsbInformation" value="Information" type="submit"></p> <p> <input name="vtNav_appName" value="" type="hidden"> <input name="vtNav_appVersion" value="" type="hidden"> <input name="vtNav_appCodeName" value="" type="hidden"> <input name="vtNav_userAgent" value="" type="hidden"></p> </form> </body> </html>
```

When 4D sends the page to a Web Browser, it looks like this:


The main features of this page are:

- It includes three Submit buttons: *vsbLogOn*, *vsbRegister* and *vsbInformation*.
- When you click Log On, the submission of the form is first processed by the JavaScript function **LogOn**. If no name is entered, the form is not even submitted to 4D, and a JavaScript alert is displayed.
- The form has a POST 4D Method as well as a Submit script (*GetBrowserInformation*) that copies the Navigator properties to the four hidden objects whose names starts with *vtNav_App*.
- It also includes the *vtUserName* object.

Let's examine the 4D method ***WWW_STD_FORM_POST*** that is called when the user clicks on one of the buttons on the HTML form.

```
 // Retrieval of value of variables
ARRAY TEXT($arrNames;0)
ARRAY TEXT($arrValues;0)
```

```
WEB GET VARIABLES($arrNames;$arrValues)
C_TEXT($user)

Case of

 // The Log On button was clicked
   :(Find in array($arrNames;"vsbLogOn")#-1)
      $user :=Find in array($arrNames;"vtUserName")
      QUERY([WWW Users];[WWW Users]UserName=$arrValues{$user})
      $0:=(Records in selection([WWW Users])>0)
      If($0)
         WWW POST EVENT("Log On";WWW Log information)
 // The WWW POST EVENT method saves the information in a database table
      Else

         $0:=WWW Register
 // The WWW Register method lets a new Web user register
      End if

 // The Register button was clicked
   :(Find in array($arrNames;"vsbRegister")#-1)
      $0:=WWW Register

 // The Information button was clicked
   :(Find in array($arrNames;"vsbInformation")#-1)
      WEB SEND FILE("userinfos.html")
End case
```

The features of this method are:

- The values of the variables *vtNav_appName*, *vtNav_appVersion*, *vtNav_appCodeName*, and *vtNav_userAgent* (bound to the HTML objects having the same names) are retrieved using the **WEB GET VARIABLES** command from HTML objects created by the *GetBrowserInformation* JavaScript script.
- Out of the *vsbLogOn*, *vsbRegister* and *vsbInformation* variables bound to the three Submit buttons, only the one corresponding to the button that was clicked will be retrieved by the **WEB GET VARIABLES** command. When the submit is performed by one of these buttons, the browser returns the value of the clicked button to 4D. This tells you which button was clicked. Note that 4D buttons in a 4D form are numeric variables. However, with HTML, all objects are text objects.

If you use a SELECT object, it is the value of the highlighted element in the object that is returned in the **WEB GET VARIABLES** command, and not the position of the element in the array as in 4D.

**WEB GET VARIABLES** always returns values of the Text type.

## Support of chunked transfer encoding

Starting with 4D v15 R3, the built-in 4D Web Server supports files uploaded in chunked transfer encoding from any Web client. Chunked transfer encoding is a data transfer mechanism specified in HTTP/1.1. It allows data to be transferred in a series of "chunks" (parts) without knowing the final data size.

**Note:** The 4D Web Server also supports chunked transfer encoding from the server to Web clients (see **WEB SEND RAW DATA**).

For more information about client-side implementation for chunked transfers, please refer to RFC7230 or the related page on Wikipedia.

## COMPILER_WEB Project Method

The **COMPILER_WEB** method, if it exists, is systematically called when the HTTP server receives a dynamic request and calls the 4D engine. This is the case, for example, when the 4D Web server receives a posted form or a URL containing the 4DCGI/ action. This method is intended to contain typing and/or variable initialization directives used during Web exchanges. It is used by the compiler when the database is compiled. The **COMPILER_WEB** method is common to all the Web forms. By default, the **COMPILER_WEB** method does not exist. You must explicitly create it.

**Web Services:** The **COMPILER_WEB** project method is called, if it exists, for each SOAP request accepted. You must use this method to declare all the 4D variables associated with incoming SOAP arguments, for all methods published as Web

Services. In fact, the use of process variables in Web Services methods requires that they be declared before the method is called. For more information on this point, refer to the description of the **SOAP DECLARATION** command.

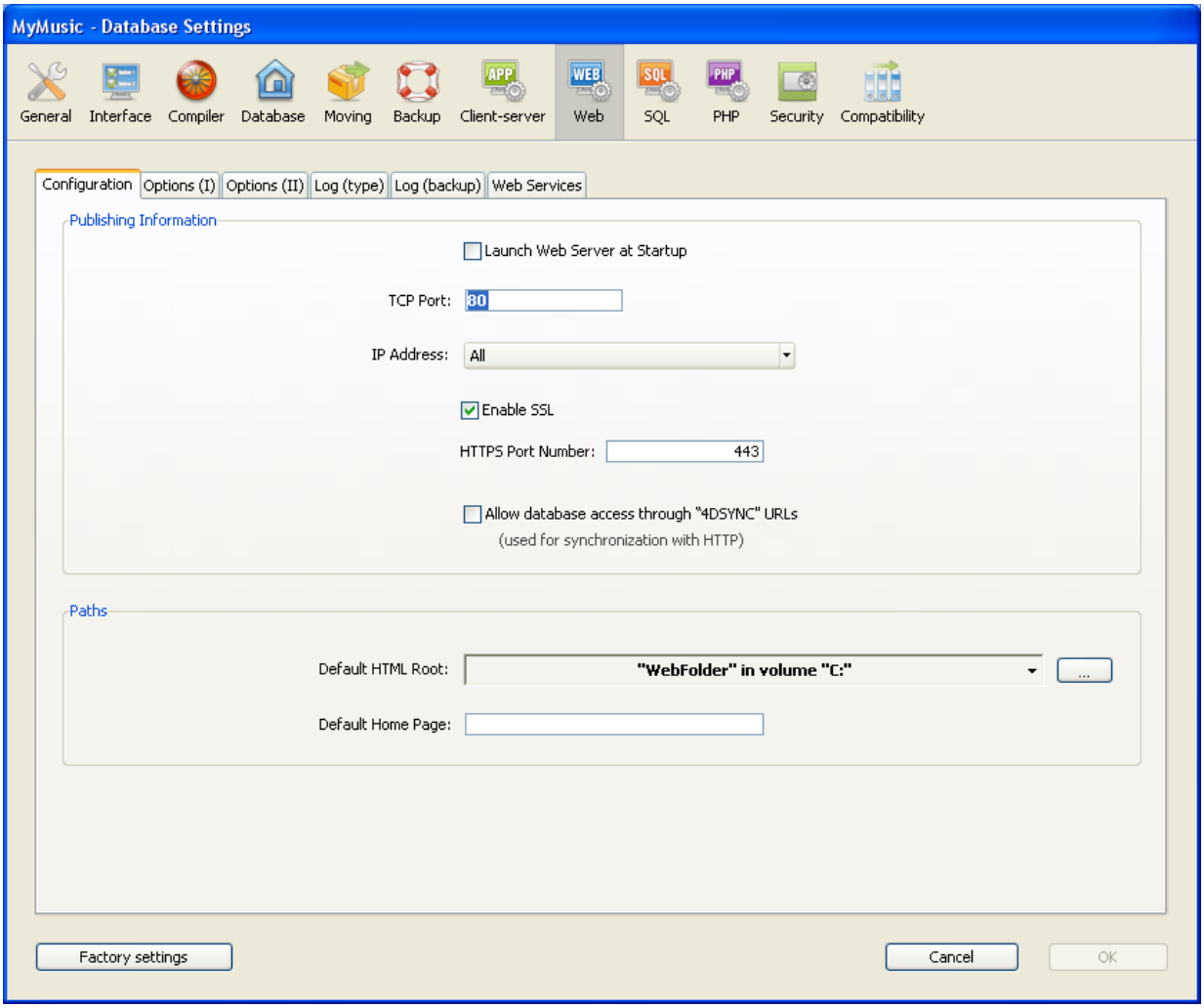
# Web Server Settings

You can configure the operation of the 4D Web server using the parameters set on the **Web** page of the Database Settings. This section describes the parameters of the **Configuration**, **Options (I)** and **(II)** tabs of this page.

- The parameters of the **Log** pages are covered in the **Information about the Web Site** section.
- The parameters of the **Web Services** page are covered in the "Design Reference" manual.

**Compatibility note:** Certain Web mechanisms found in previous versions of 4D are now considered obsolete (for example, "Do not remove “/” on unknown URLs"). For compatibility's sake, these mechanisms can still be used in converted databases. In this case, you can display them and if necessary disable them on the **Compatibility** page of the Database Settings.

## Configuration page




### Launch Web Server at Startup

Indicates whether the Web server will be launched on startup of the 4D application. This option is described in the **Web server configuration and connection management** section.

### TCP port number

By default, 4D publishes a Web database on the regular Web TCP Port, which is port 80. If that port is already used by another Web service, you need to change the TCP Port used by 4D for this database. Modifying the TCP port allows you to start the 4D Web server under Mac OS X without being the root user of the machine (see **Web server configuration and connection management** section).

To do so, go to the **TCP Port** enterable area and indicate an appropriate value (a TCP port not already used by another TCP/IP service running on the same machine).

**Note:** If you specify 0, 4D will use the default TCP port number 80.

From a Web browser, you need to include that non-default TCP port number into the address you enter for connecting to the Web database. The address must have a suffix consisting of a colon followed by the port number. For example, if you are using the TCP port number 8080, you will specify “123.4.567.89:8080”.

**WARNING:** If you use TCP port numbers other than the default numbers (80 for standard mode and 443 for SLL mode), be careful not to use port numbers that are defaults for other services that you might want to use simultaneously. For example, if you also plan to use the FTP protocol on your Web server machine, do not use the TCP port 20 and 21, which are the default ports for that protocol (unless you know what you are doing). To find out the standard assignment of TCP port numbers, refer to the **Appendix B, TCP Port Numbers** section in the documentation of the 4D Internet Commands. Ports numbers below 256 are reserved for well known services and ports numbers from 256 to 1024 are reserved for specific services originated on the UNIX platforms. For maximum security, specify a port number beyond these intervals, for example in the 2000's or 3000's.

### Defining the IP Address for the HTTP Requests

You can define the IP address on which the Web server must receive HTTP requests.

**Note:** Starting with 4D v14, the HTTP server automatically supports IPv6 address notation when the **All** option is selected in the "IP Address" list. For more information, refer to **Support of IPv6**.

By default, the server responds to all IP addresses (**All** option).

The drop-down list automatically lists all available IP addresses on the machine. When you select a specific address, the server only responds to requests sent to this address. This feature is for 4D Web Servers located on machines with multiple TCP/IP addresses. It is, for example, frequently the case of most Internet host providers. Implementing such a MultiHoming system requires specific configurations on the Web server machine:

- **Installing secondary IP addresses on Mac OS**

To configure a MultiHoming system on Mac OS:

1. Open the **TCP/IP** Control Panel.
2. Select the **Manually** option from the **Configuration** pop up menu.
3. Create a text file called "Secondary IP Addresses" and save it in the Preferences subfolder of your System folder.
   Each line of the "Secondary IP Addresses" file should contain a secondary IP address and an optional subnet mask and router address for the secondary IP address.

Please check the Apple documentation for more information.

- **Installing secondary IP addresses on Windows**

To configure a MultiHoming system on Windows:

1. Select the following sequences of commands (or their equivalents according to your version of Windows):
   **Start** menu > **Control Panel** > **Network and Internet Connections** > **Network connections** > **Local Area Connection** (Properties) > **Internet Protocol (TCP/IP)** > **Properties** button > **Advanced...** button. The "Advanced TCP/IP Settings" dialog is displayed.
2. Click the **Add...**. button in the "IP Addresses" area, and add additional IP addresses.

You can define up to 5 different IP addresses. You may need to consult your systems administrator in order to do so.

### Enable SSL

Indicates whether or not the Web server will accept secure connections. This option is described in the **Using TLS Protocol** section.

### HTTPS Port Number

Allows you to modify the TCP/IP port number used by the Web server for secured HTTP connections over SSL (HTTPS protocol). By default, the HTTPS port number is set to 443 (standard value).
You may consider changing this port number for two main reasons:

- for security reasons — hacker attacks against Web servers are generally concentrated on standard TCP ports (80 and 443).
- under Mac OS X, in order to allow “standard” users to launch the Web server in a secured mode — under Mac OS X, the use of TCP/IP ports reserved for Web publications (0 to 1023) requires specific access privileges: only the root user can

launch an application using these ports. In order for standard users to be able to launch the Web server, one solution is to modify the TCP/IP port number (see the **Web server configuration and connection management** section). You can pass any valid value (in order to avoid access restrictions under Mac OS X, you should pass a value greater than 1023). For more information about TCP port numbers, refer to the “TCP port number” paragraph above.

### Allow database access through 4DSYNC URLs

This option controls support of HTTP synchronization requests containing /4DSYNC URLs. It is covered in the **Connection Security** section.

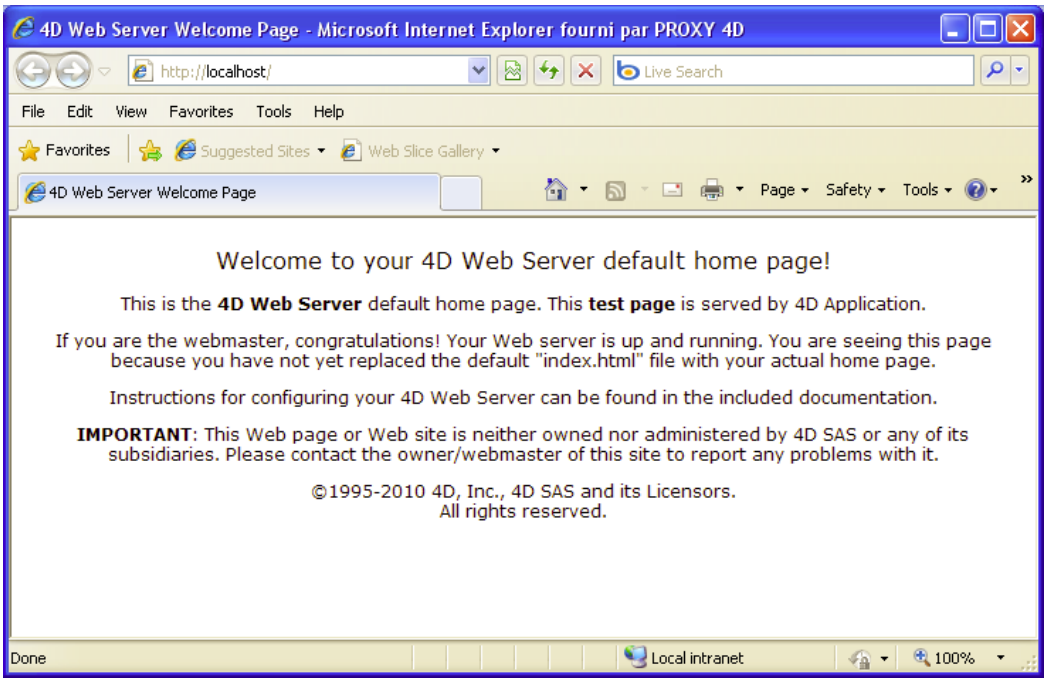
### Default HTML Root

Allows you to define the default location of the Web site files and to indicate the hierarchical level on the disk above which the files will not be accessible. This option is described in the **Connection Security** section**.**

### Defining a Default Home Page

You can designate a default home page for all the browsers that connect to the database. This page can be static or semi-dynamic.

By default, when the Web server is launched for the first time, 4D creates a home page named “index.html” and puts it in the HTML root folder. If you do not modify this configuration, any browser connecting to the Web server will obtain the following page:




To modify the default home page, simply replace it in the database root folder with your own “index.html” page or enter the relative access path of the page that you want to define in the “Default Home Page” entry area.
The access path must be set up in relation to the default HTML root folder.
In order to ensure multi-platform compatibility of your databases, the 4D Web server uses particular writing conventions to define access paths. The syntax rules are as follows:

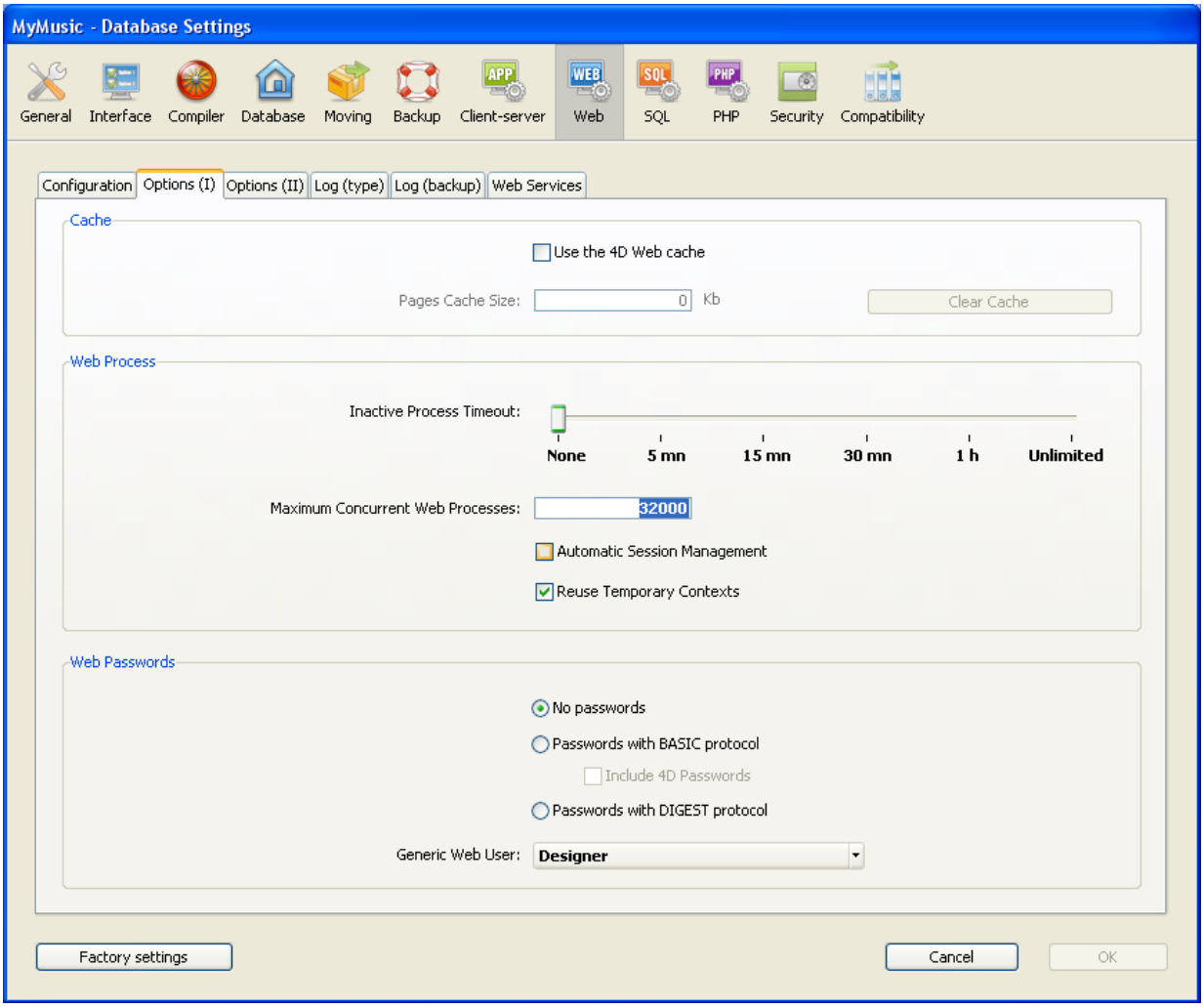
- folders are separated by a slash (“/”)
- the access path must not end with a slash (“/”)
- to “go up” one level in the folder hierarchy, enter “..” (two periods) before the folder name
- the access path must not start with a slash (“/”)

For example, if you want the default home page to be “MyHome.htm”, and it is located in the “Web” folder (itself located in the default HTML root folder of the database), enter “Web/MyHome.htm”.

**Note:** You can also define a default home page for each Web process by using the routine **WEB SET HOME PAGE**.

If you do not specify a default custom home page, the **On Web Connection Database Method** is called. It is up to you to process the request procedurally.

## Options (I) Page




## Cache for Static Pages

The 4D Web Server has a cache that allows you to load static pages, GIF images, JPEG images (<512 kb) and style sheets (.css files) in memory, as they are requested.
Using the cache allows you to significantly increase the Web server's performance when sending static pages.

The cache is shared between all the Web processes. You can set the size of the cache in the Preferences. By default, the cache of the static pages is not enabled. To activate it, simply check the **Use the 4D Web cache** option.

You can modify the size of the cache in the **Pages Cache Size** area. The value you set depends on the number and size of your Web site's static pages, as well as the resources that the host machines has at its disposal.

**Note:** While using your Web database, you can check the performance of the cache by using the routine **WEB GET STATISTICS**. If, for example, you notice that the cache's rate of use is close to 100%, you may want to consider increasing the size that has been allocated to it.
The /4DSTATS and /4DHTMLSTATS URLs allow you to also obtain information about the cache's state. Please refer to **Information about the Web Site**.

Once the cache has been enabled, the 4D Web server looks for the page requested by the browser first in the cache. If it finds the page, it sends it immediately. If not, 4D loads the page from disk and places it in the cache.
When the cache is full and additional space is required, 4D "unloads" the oldest pages first, among the least demanded ones.

### *Clearing the Cache*

At any moment, you can clear the cache of the pages and images that it contains (if, for example, you have modified a static page and you want to reload it in the cache).
To do so, you just have to click on the **Clear Cache** button. The cache is then immediately cleared.

**Note:** You can also use the special URL **/4DCACHECLEAR**.

## Inactive Process Timeout

Allows you to set the maximum timeout before closing for inactive Web processes on the server.

## Maximum Concurrent Web Processes

This option indicates the strictly high limit of **Maximum Concurrent Web Processes** of any type (standard Web processes or belonging to the"pool of processes") that can be simultaneously open on the server. This parameter allows prevention of

4D Server saturation as the result of massive number of requests.

By default, this value is 32000. You can set the number anywhere between 10 and 32000.

When the maximum number of concurrent Web processes (minus one) is reached, 4D no longer creates new processes and sends the following message “Server unavailable” (status HTTP 503 – Service Unavailable) for each new request.

**Note:** You can also set the maximum number of Web using the **WEB SET OPTION** command.

***About the Pool of Web Processes***

The “pool” of Web processes allows increasing the reactivity of the Web server. This reserve is sized by a minimum (0 by default) and a maximum (10 by default) of processes to recycle. These processes can be modified using the **SET DATABASE PARAMETER** command. Once the maximum number of Web processes has been changed, if this number is inferior to the superior limit in the “pool”, this limit is lowered to the maximum number of Web processes.

***How to determine the right value?***

In theory, the maximum number of Web processes is the result of the following formula: Available memory/Web process stack size(*).
Another solution is to visualize the information on Web processes displayed in the Runtime Explorer: the current number of Web processes and the maximum number reached since the Web server boot are indicated.

(*) The stack size allocated by 4D for a Web process is around 512 KB for 64-bit versions and around 256 KB for 32-bit versions (indicative values, which may vary based on context).

### Automatic Session Management

Enables or disables the internal mechanism for automatic handling of user sessions by the 4D HTTP server. This mechanism is described in the **Web Sessions Management** section.

By default, this mechanism is enabled in databases created with 4D v13 and later versions. However, for compatibility reasons, it is disabled in databases converted from previous versions of 4D. You must enable it explicitly in order to benefit from this functionality.

When this option is checked, the "Reuse Temporary Contexts" option is automatically checked (and locked).

### Reuse temporary contexts (in remote mode)

Allows you to optimize the operation of the 4D Web server in remote mode by reusing Web processes created for processing previous Web requests. In fact, the Web server of 4D needs a specific Web process for the handling of each Web request; in remote mode, when necessary, this process connects to the 4D Server machine in order to access the data and database engine. It thus generates a temporary context using its own variables, selections, etc. Once the request has been dealt with, this process is killed.

When the **Reuse Temporary Contexts** option is checked, in remote mode 4D maintains the specific Web processes and reuses them for subsequent requests. By removing the process creation stage, Web server performance is improved.

In return, you must make sure in this case to systematically initialize the variables used in 4D methods in order to avoid getting incorrect results. Similarly, it is necessary to erase any current selections or records defined during the previous request.

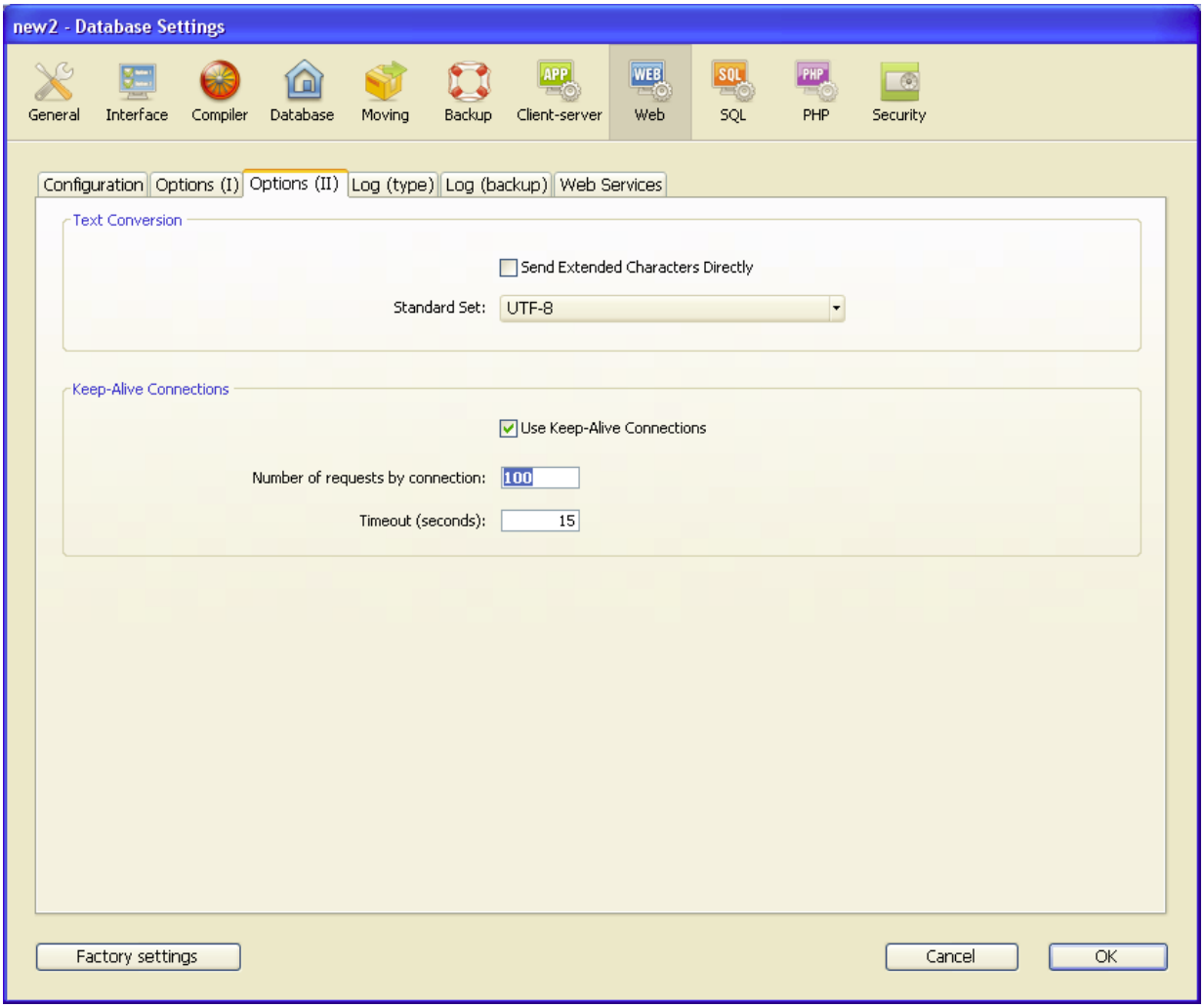
**Notes:**

- This option is checked (and locked) automatically when the **Automatic Session Management** option is checked. In fact, the session management mechanism is actually based on the principle of recycling Web processes: each session uses the same process that is maintained during the lifespan of the session. However, note that session processes cannot be "shared" between different sessions: once the session is over, the process is automatically killed (and not reused). It is therefore unnecessary to reset the selections or variables in this case.
- This option only has an effect with a 4D Web server in remote mode. With a 4D in local mode, all Web processes (other than session processes) are killed after their use.

### “Passwords” area

Configuration of Web site access protection using passwords. This option is described in the **Connection Security** section.

## Options (II) Page




### Directly Sending Extended Characters

By default, the 4D Web server converts the extended characters in the dynamic and static Web pages according to HTML standards before sending them. They are then interpreted by the browsers.
You can set the Web server so that the extended characters are sent “as is”, without converting them into HTML entities. This option has shown a speed increase on most foreign operating systems (especially the Japanese system).

To do this, check the **Send Extended Characters Directly** option.

### Standard Sets

The **Standard Set** drop-down list allows you to define the set of characters to be used by the 4D Web server. By default, the character set is UTF-8.

**Note:** This setting is also used for generating Quick Reports in HTML format (see **Executing a quick report**).

### Keep-Alive Connections

The Web server of 4D can use keep-alive connections. The keep-alive option allows you to maintain a single open TCP connection for the set of exchanges between the Web browser and the server to save system resources and to optimize transfers.

The **Use Keep-Alive Connections** option enables or disables keep-alive TCP connections for the Web server. This option is enabled by default. In most cases, it is advisable to keep this option check since it accelerates the exchanges. If the Web browser does not support connection keep alive, the 4D Web server automatically switches to HTTP/1.0.

The 4D Web server keep-alive function concerns all TCP/IP connections (HTTP, HTTPS). Note however that keep-alive connections are not always used for all 4D Web processes.

In some cases, other optimized internal functions may be invoked. Keep-alive connections are useful mainly for static pages.

Two options allow you to set how the keep-alive connections work:

- **Number of requests by connection**: Allows you to set the maximum number of requests and responses able to travel over a connection keep alive. Limiting the number of requests per connection allows you to prevent server flooding due to a large number of incoming requests (a technique used by hackers).

The default value (100) can be increased or decreased depending on the resources of the machine hosting the 4D Web server.

- **Timeout**: This value defines the maximum wait period (in seconds) during which the Web server maintains an open TCP connection without receiving any requests from the Web browser. Once this period is over, the server closes the connection.
  If the Web browser sends a request after the connection is closed, a new TCP connection is automatically created. This operation is not visible for the user.

# Using preemptive Web processes

The built-in Web Server of 4D 64-bit for Windows and OS X allows you to take full advantage of multi-core computers by using preemptive Web processes in your compiled applications. You can configure your Web-related code, including 4D tags and Web database methods, to run simultaneously on as many cores as possible.

For more information on preemptive process feature in 4D, please refer to the **Preemptive 4D processes** section.

## Availability of preemptive mode for Web processes

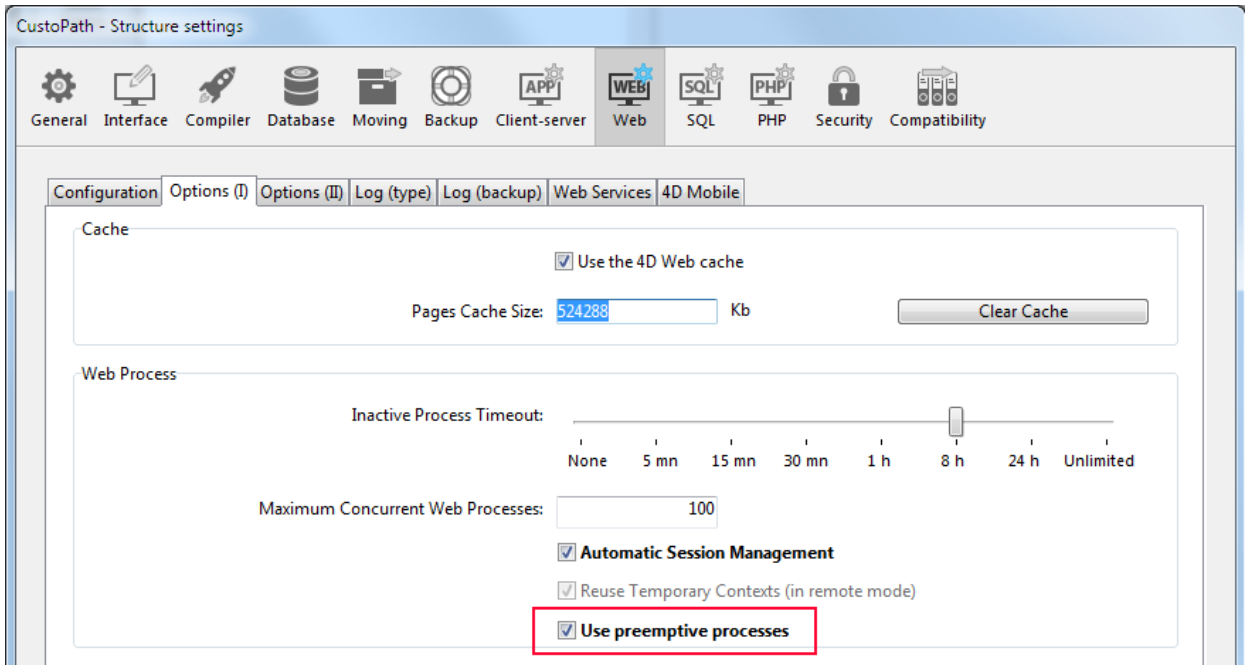
The use of preemptive mode for Web processes is only available in the following contexts:

- use of a 64-bit version of 4D
- use of 4D Server or 4D local mode (4D in remote mode does not support preemptive mode)
- use of a compiled database
- **Use preemptive processes** database setting checked (see below)
- all Web-related database methods and project methods are confirmed thread-safe by 4D Compiler

If any requirement is missing, the Web Server will use cooperative processes.

## Enabling the preemptive mode for the Web Server

To enable the preemptive mode for the Web Server code of your application, you must check the **Use preemptive processes** option on the "Web/Options (I)" page of the Database Settings dialog box:




When this option is checked, the 4D compiler will automatically evaluate the thread-safety property of each piece of Web-related code (see below) and return errors in case of incompatibility.

## Writing thread-safe Web Server code

All 4D code executed by the Web Server must be thread-safe if you want your Web processes to be run in preemptive mode. When the **Use preemptive processes** option is checked in the Database Settings dialog box, the following parts of the application will be automatically evaluated by the 4D compiler:

- All Web-related database methods:
  - **On Web Authentication database method**
  - **On Web Connection database method**

    - **On Web Close Process database method**
    - **On 4D Mobile Authentication database method**
- The ***compiler_web*** project method (regardless of its actual "Execution mode" property);
- Basically any code processed by the **PROCESS 4D TAGS** command in the Web context, for example through .shtml pages.
- Any project method with the "Available through 4D HTML tags and URLS (4DACTION...)" attribute
- Triggers for tables with "Expose with 4D Mobile Service" attribute
- Project methods available through 4D Mobile ("4D Mobile" property checked)

For each of these methods and code parts, the compiler will check if the thread-safety rules are respected, and will return errors in case of issues. For more information about thread-safety rules, please refer to the **Writing a thread-safe method** paragraph.

## Thread-safety of 4D commands and Web URLs

Starting with 4D v16, most of the Web-related 4D commands, database methods and URLs are thread-safe and can be used in preemptive mode:

### 4D commands

All 4D Web-related commands are thread-safe, i.e.:

- all commands from the "**Web Server**" theme,
- all commands from the "**HTTP Client**" theme.

### Database methods

The following database methods are thread-safe and can be used in preemptive mode:

- **On Web Authentication database method**
- **On Web Connection database method**
- **On Web Close Process database method**
- **On 4D Mobile Authentication database method**

Of course, the code executed by these methods must also be thread-safe.

### Web Server URLs

The following 4D Web Server URLs are thread-safe and can be used in preemptive mode:

- 4daction/ (the called project method must also be thread-safe)
- 4dcgi/ (the called database methods must also be thread-safe)
- 4dscript/ (deprecated as URL, used as a tag)
- 4dwebtest/
- 4dblank/
- 4dstats/
- 4dhtmlstats/
- 4dcacheclear/
- rest/
- 4dimgfield/ (generated by **PROCESS 4D TAGS** for web request on picture fields)
- 4dimg/ (generated by **PROCESS 4D TAGS** for web request on picture variables)

The following 4D Web Server URLs are not thread-safe and not supported in preemptive mode:

- 4dsync
- 4dsqlauth (deprecated, used for Flex 1.1)

## Preemptive Web process icon

Both the Runtime Explorer and the 4D Server administration window display a specific icon for preemptive Web processes:

| Process type | Icon |
| --- | --- |
| Preemptive Web method |  |

## Information about the Web Site

4D allows you to obtain information about the functioning of your 4D Web site.

- You can control the site by using particular URLs (*/4DSTATS*, */4DHTMLSTATS*, */4DCACHECLEAR* and */4DWEBTEST*).
- You can generate a log of all the requests.
- You can obtain information regarding the Web Server in the Watch page of the Runtime Explorer window.

**Note:** For security reasons, starting with 4D v15 R2, the HTTP TRACE method is disabled by default in the 4D Web Server. This means that when an HTTP TRACE request is received, the 4D Web Server now returns a 405 error ("method not allowed"). If you want to explicitly enable the HTTP TRACE method, you must use the Web HTTP TRACE option with the **WEB SET OPTION** command.

### Web Server Management URLs

4D Web Server accepts four particular URLs: */4DSTATS*, */4DHTMLSTATS*, */4DCACHECLEAR* and */4DWEBTEST*.

- */4DSTATS*, */4DHTMLSTATS* and */4DCACHECLEAR* are only available to the Designer and Administrator of the database. If the database's 4D password system has not been activated, these URLs are available to all the users.
- */4DWEBTEST* is always available.

#### /4DSTATS

The */4DSTATS* URL returns several items of information in an HTML table (displayable in a browser):

- **Cache Current Size**: Current size of Web server cache (in bytes)
- **Cache Max Size**: Maximum size of cache (in bytes)
- **Cached Object Max Size**: Maximum size of each object in the cache (in bytes)
- **Cache Use**: Percentage of cache used
- **Cached Objects**: Number of objects (pages, picture files, etc.) found in the cache.

(*) For more information about the cache of static pages and pictures, please refer to the **Web Server Settings** section.

This information can allow you to check the functioning of your server and eventually adapt the corresponding parameters.

**Note:** The command **WEB GET STATISTICS** allows you to also obtain information about how the cache is being used for static pages.

#### /4DHTMLSTATS

The */4DHTMLSTATS* URL returns, also as an HTML table, the same information as the */4DSTATS* URL. The difference is that the **Cached Objects** field only counts HTML pages (without counting picture files). Moreover, this URL returns the **Filtered Objects** field.

- **Filtered Objects**: Number of objects in cache not counted by URL, in particular, pictures.

#### /4DCACHECLEAR

The */4DCACHECLEAR* URL immediately clears the cache of the static pages and images. It allows you to therefore "force" the update of the pages that have been modified.

#### /4DWEBTEST

The */4DWEBTEST* URL is designed to check the Web Server status. When this URL is called, 4D returns a text file only with the following HTTP fields filled:

- **Date:** current date at the RFC 822 format
  For example: "Mon, 16 Jan 2012 13:12:50 GMT"
- **Server:** 4D_version/internal version number
  For example: "4D_v12/12.3.0"
- **User-Agent:** name and version @ IP client address
  For example: "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:9.0.1) Gecko/20100101 Firefox/9.0.1 @ 127.0.0.1

## Connection Log File

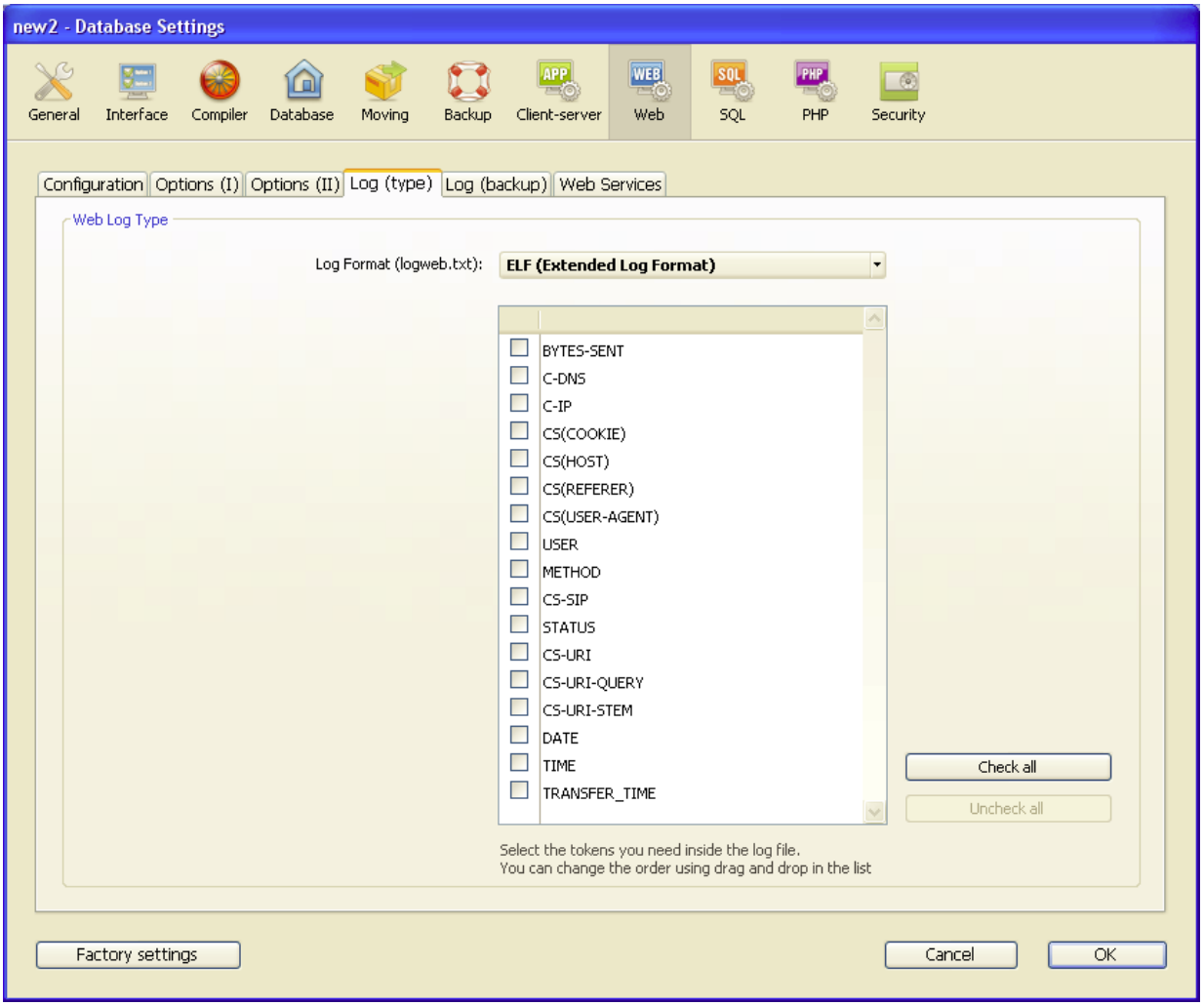
4D allows you to obtain a log of requests.

This file is named “logweb.txt” and is automatically located:

- with 4D in local mode and 4D Server, in the Logs folder located next to the database structure file.
- with 4D in remote mode, in the Logs subfolder of the 4D client database folder (cache folder).

### Activation and Format

The activation and configuration of the log file contents is carried out in the Database Settings on the **Web/Log (type)** page:




**Note:** The activation and deactivation of the log file of requests can also be carried out by programming using the **SET DATABASE PARAMETER** (4D v12) or **WEB SET OPTION** (4D v13 and higher) commands.

The log format menu provides the following options:

- **No Log File**: When this option is selected, 4D will not generate a log file of requests.
- **CLF (Common Log Format)**: When this option is selected, the log of requests is generated in CLF format. With the CLF format, each line of the file represents a request, such as:
  *host rfc931 user [DD/MMM/YYYY:HH:MM:SS] "request" state length*
  Each field is separated by a space and each line ends by the CR/LF sequence (character 13, character 10).
  - host: IP address of the client (ex. 192.100.100.10)
  - rfc931: information not generated by 4D, it’s always - (a minus sign)
  - user: user name as it is authenticated, or else it is - (a minus sign). If the user name contains spaces, they will be replaced by _ (an underscore).
  - DD: day, MMM: a 3-letter abbreviation for the month name (Jan, Feb,...), YYYY: year, HH: hour, MM: minutes, SS: seconds

  The date and time are local to the server.

  - request: request sent by the client (ex. GET /index.htm HTTP/1.0)
  - state: reply given by the server.

- length: size of the data returned (except the HTTP header) or 0.

**Note:** For performance reasons, the operations are saved in a memory buffer in packets of 1Kb before being written to disk. The operations are also written to disk if no request has been sent every 5 seconds.
The possible values of state are as follows:
200: OK
204: No contents
302: Redirection
304: Not modified
400: Incorrect request
401: Authentication required
404: Not found
500: Internal error
The CLF format cannot be customized.

- **DLF (Combined Log Format)**: When this option is selected, the request log is generated in DLF format. DLF format is similar to CLF format and uses exactly the same structure. It simply adds two additional HTTP fields at the end of each request: Referer and User-agent.
  - Referer: Contains the URL of the page pointing to the requested document.
  - User-agent: Contains the name and version of the browser or software of the client at the origin of the request.

  The DLF format cannot be customized.

- **ELF (Extended Log Format)**: When this option is selected, the request log is generated in ELF format. The ELF format is very widespread in the world of HTTP browsers. It can be used to build sophisticated logs that meet specific needs. For this reason, the ELF format can be customized: it is possible to choose the fields to be recorded as well as their order of insertion into the file.

- **WLF (WebStar Log Format)**: When this option is selected, the request log is generated in WLF format. WLF format was developed specifically for the 4D WebSTAR server. It is similar to the ELF format, with only a few additional fields. Like the ELF format, it can be customized.

**Configuring the fields**
When you choose the ELF (Extended Log Format) or WLF (WebStar Log Format) format, the “Weg Log Token Selection” area displays the fields available for the chosen format. You will need to select each field to be included in the log. To do so, use the arrow buttons or simply drag and drop the desired fields into the “Selected Tokens” area.

**Note**: You cannot select the same field twice.

The following table lists the fields available for each format (in alphabetical order) and describes its contents:

| Field | ELF | WLF | Value |
|---|---|---|---|
| BYTES_RECEIVED | | X | Number of bytes received by the server |
| BYTES_SENT | X | X | Number of bytes sent by the server to the client |
| C_DNS | X | X | IP address of the DNS (ELF: field identical to the C_IP field) |
| C_IP | X | X | IP address of the client (for example 192.100.100.10) |
| CONNECTION_ID | | X | Connection ID number |
| CS(COOKIE) | X | X | Information about cookies contained in the HTTP request |
| CS(HOST) | X | X | Host field of the HTTP request |
| CS(REFERER) | X | X | URL of the page pointing to the requested document |
| CS(USER_AGENT) | X | X | Information about the software and operating system of the client |
| CS_SIP | X | X | IP address of the server |
| CS_URI | X | X | URI on which the request is made |
| CS_URI_QUERY | X | X | Request query parameters |
| CS_URI_STEM | X | X | Part of request without query parameters |
| DATE | X | X | DD: day, MMM: 3-letter abbreviation for month (Jan, Feb, etc.), YYYY: year |
| METHOD | X | X | HTTP method used for the request sent to the server |
| PATH_ARGS | | X | CGI parameters: string located after the “$” character |
| STATUS | X | X | Reply provided by the server |
| TIME | X | X | HH: hour, MM: minutes, SS: seconds |
| TRANSFER_TIME | X | X | Time requested by server to generate the reply |
| USER | X | X | User name if authenticated; otherwise - (minus sign).<br>If the user name contains spaces, they are replaced by _ (underlines) |
| URL | | X | URL requested by the client |

**Note:** Dates and times are given in GMT.

### Periodicity of Backups

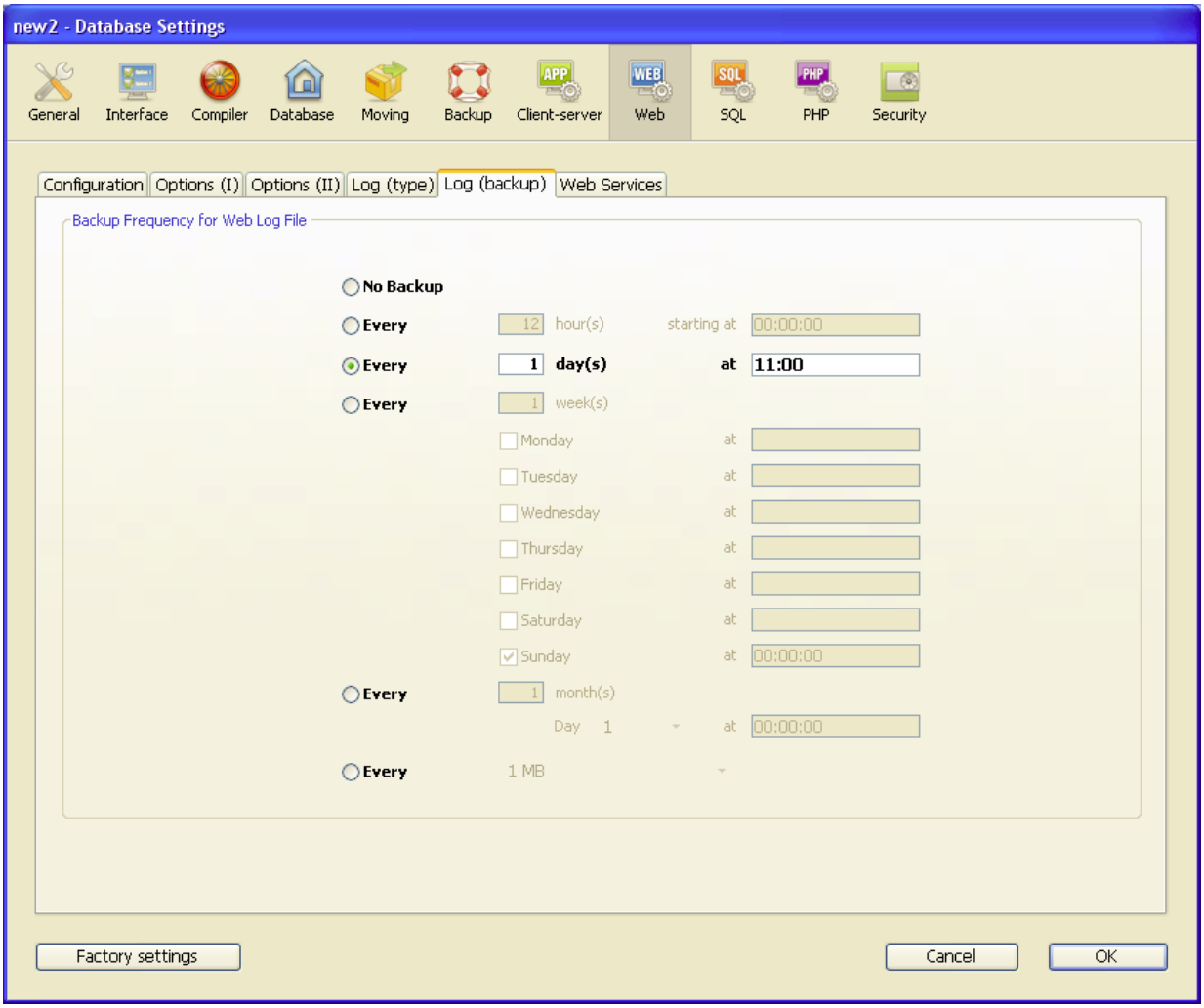
Since a Web log file can become considerably large, it is possible to set up an automatic archiving mechanism. The triggering of a backup can be based on a certain period of time (expressed in hours, days, week or months), or based on the file size; when the set deadline (or file size) is reached, 4D automatically closes and archives the current log file and creates a new one.

When the Web log file backup is triggered, the log file is archived in a folder named “Logweb Archives,” which is created at the same level as the logweb.txt file (that is, next to the database structure file).

The archived ile is renamed based on the following example: “DYYYY_MM_DD_Thh_mm_ss.txt.” For instance, for a file archived on September 4, 2006 at 3:50 p.m. and 7 seconds: “D2006_09_04_T15_50_07.txt.”

### Backup Parameters

The automatic backup parameters for the request log are set on the **Web/Log (backup)** page of the Database Settings:




First you must choose the frequency (days, weeks, etc.) or the file size limit criterion by clicking on the corresponding radio button. You must then specify the precise moment of the backup if necessary.

- **No Backup**: The scheduled backup function is deactivated.
- **Every X hour(s)**: This option is used to program backups on an hourly basis. You can enter a value between 1 and 24 .
  - **starting at**: Used to set the time at which the first back up will begin.
- **Every X day(s) at X**: This option is used to program backups on a daily basis. Enter 1 if you want to perform a daily backup. When this option is checked, you must indicate the time when the backup must be started.
- **Every X week(s), day at X**: This option is used to program backups on a weekly basis. Enter 1 if you want to perform a weekly backup. When this option is checked, you must indicate the day(s) of the week and the time when each backup must be started. You can select several days of the week if desired. For example, you can use this option to set two weekly backups: one on Wednesdays and one on Fridays.
- **Every X month(s), Xth day at X**: This option is used to program backups on a monthly basis. Enter 1 if you want to perform a monthly backup. When this option is checked, you must indicate the day of the month and the time when the backup must be started.
- **Every X MB**: This option is used to program backups based on the size of the current request log file. A backup is automatically triggered when the file reaches the set size. You can set a size limit of 1, 10, 100 or 1000 MB.

**Note:** In the case of scheduled backups, if the Web server was not launched when the backup was scheduled to occur, on the next startup 4D considers the backup as failed and applies the appropriate settings, set via the Database Settings.

## Runtime Explorer Information

The **Watch** page (“Web” heading) in the Runtime Explorer displays information related to the Web Server, more particularly:

- **Web Cache Usage**: indicates the number of pages present in the Web cache as well as its use percentage. This information is only available if the Web server is active and if the cache size is greater than 0.
- **Web Server Elapsed Time**: indicates the duration of use (in hours:minutes:seconds format) of the Web server. This information is only available if the Web server is active.
- **Web Hits Count**: indicates the total number of HTTP requests received since the Web server boot, as well as an instantaneous number of requests per second (measure taken between two Runtime Explorer updates). This

information is only available if the Web server is active.

**Note:** For more information about the Runtime Explorer, refer to 4D *Design Reference* manual.

# Using TLS Protocol

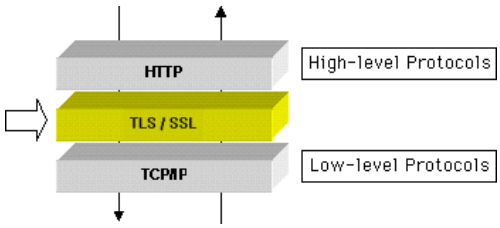
The 4D Web server can communicate in secured mode through the TLS (Transport Layer Security) protocol -- the successor of the SSL (Secured Socket Layer) protocol.

## TLS Protocol Definition

The TLS protocol (successor of SSL) has been designed to secure data exchanges between two applications —mainly between a Web server and a browser. This protocol is widely used and is compatible with most Web browsers.

At the network level, the security protocol is inserted between the TCP/IP layer (low level) and the HTTP high level protocol. It has been designed mainly to work with HTTP.

Network configuration using TSL:




**Note:** The TLS protocol can also be used to secure standard 4D Server client/server connections as well as SQL server connections. For more information, refer to the section **Encrypting Client/Server Connections** in the 4D Server Reference manual as well as the **Configuration of 4D SQL Server** section in the SQL Reference manual.

The TLS protocol is designed to authenticate the sender and receiver and to guarantee the confidentiality and integrity of the exchanged information:

- **Authentication**: The sender and receiver identities are confirmed.
- **Confidentiality**: The sent data is encrypted so that no third person can understand the message.
- **Integrity**: The received data has not been changed, by accident or malevolently.

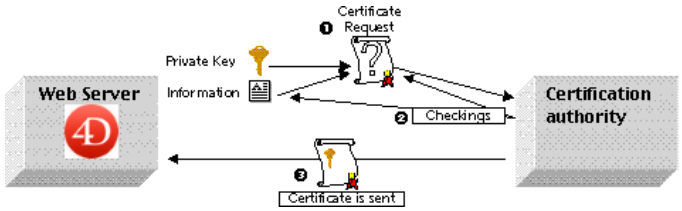
TLS uses a public key encryption technique based on a pair of asymmetric keys for encryption and decryption: a public key and a private key.
The private key is used to encrypt data. The sender (the web site) does not give it to anyone. The public key is used to decrypt the information and is sent to the receivers (Web browsers) through a certificate. When using TLS with the Internet, the certificate is delivered through a certification authority, such as Verisign®. The Web site pays the Certificate Authority to deliver a certificate which guaranties the server authentication and contains the public key allowing to exchange data in a secured mode.

**Note**: For more information on the encryption method and the public and private key issues, refer to the **ENCRYPT BLOB** command description.

## How to get a certificate?

A server working in secured mode means that you need a digital certificate from a certification authority. This certificate contains various information such as the site ID as well as the public key used to communicate with the site. This certificate is transmitted to the clients (Web browsers) connecting to this site. Once the certificate has been identified and accepted, the communication is made in secured mode.

**Note:** A browser authorizes only the certificates issued by a certification authority referenced in its properties.




The certification authority is chosen according to several criteria. If the certification authority is well known, the certificate will be authorized by many browsers, however the price to pay will be expensive.

To get a digital certificate:

1. Generate a private key using the **GENERATE ENCRYPTION KEYPAIR** command.

**Warning:** For security reasons, the private key should always be kept secret. Actually, it should always remain with the server machine. For the Web server, the **Key.pem** file must be placed in the Database structure folder.

2. Use the **GENERATE CERTIFICATE REQUEST** command to issue a certificate request.

3. Send the certificate request to the chosen certificate authority.
To fill in a certificate request, you might need to contact the certification authority. The certification authority checks that the information transmitted are correct. The certificate request is generated in a BLOB using the PKCS format encoded in base64 (PEM format). This principle allows you to copy and paste the keys as text and to send them via E-mail without modifying the key content. For example, you can save the BLOB containing the certificate request in a text document (using the **BLOB TO DOCUMENT** command), then open and copy and paste its content in a mail or a Web form to be sent to the certification authority.

4. Once you get your certificate, create a text file named “cert.pem” and paste the contents of the certificate into it.
You can receive a certificate in different ways (usually by E-mail or HTML form). 4D accepts all platform-related text formats for certificates (OS X, PC, Linux...). However, the certificate must be in PEM format, i.e. PKCS encoded in base64.

**Note:** CR line-ending characters are not supported on their own; you must use CRLF or LF.

5. Place the “cert.pem” file in the correct location. For the Web server, this is the folder containing the database structure.
The Web server can now work in a secured mode. A certificate is valid between 6 months to a year.

## TLS installation and activation within 4D

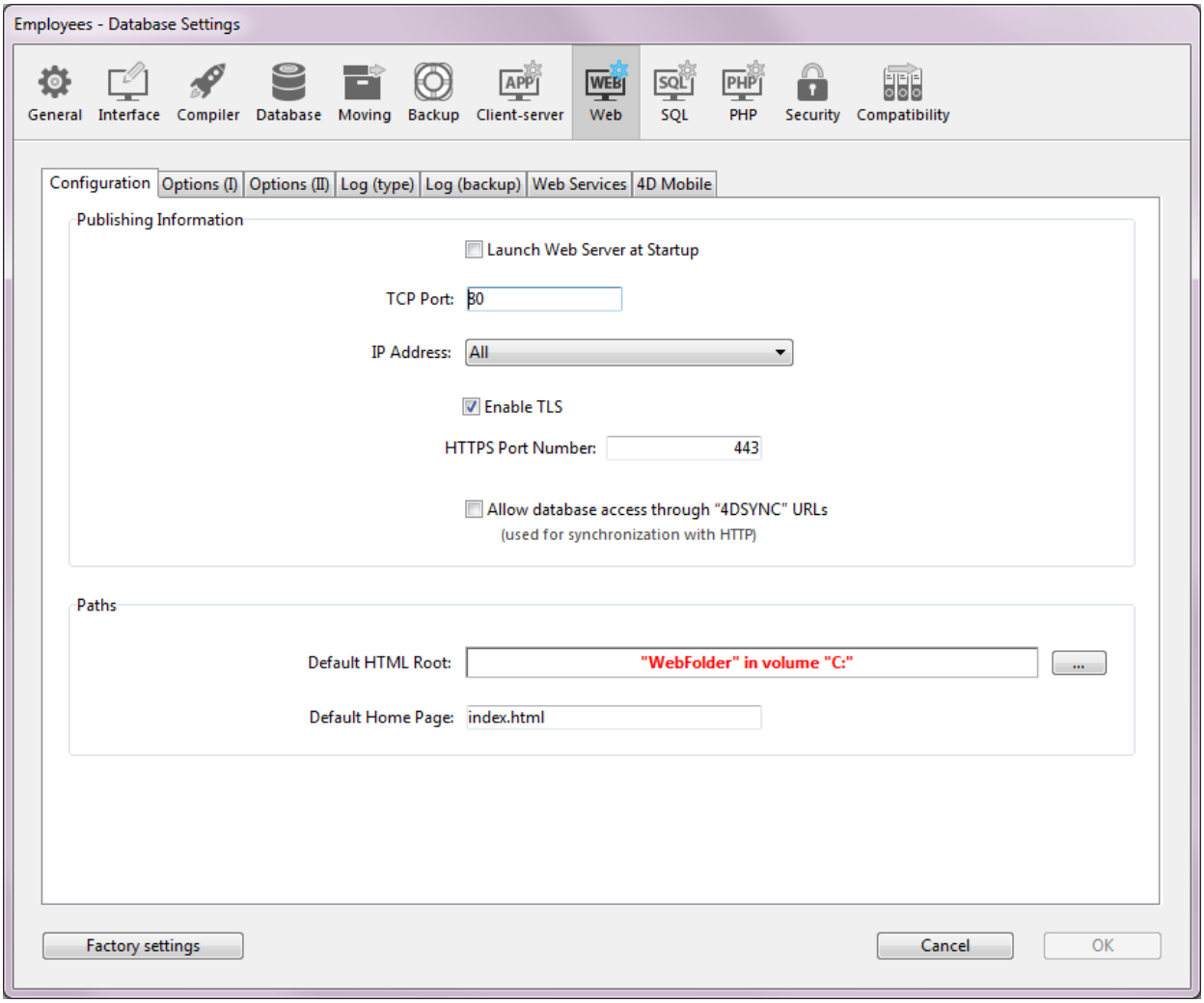
If you want to use the TLS protocol with the 4D Web server, the following components should be installed on the server, at different locations:

- *4DSLI.DLL* (Windows) or *4DSLI.bundle* (Mac OS): Secured Layer Interface dedicated to the TLS management.
  This file is installed by default, it is placed:
  - Under Windows, next to the executable file of the 4D or 4D Server application
  - Under Mac OS, in the *Native Components* subfolder of the 4D or 4D Server package.

- *key.pem* (document containing the private encryption key) and *cert.pem* (document containing the certificate):
  - with 4D in local mode or 4D Server, these files must be placed next to the database structure file
  - with 4D in remote mode, these files must be located in the local resources folder of the database on the remote machine (for more information about the location of this folder, refer to the 4D Client Database Folder paragraph in the description of the **Get 4D folder** command). Note that you must copy these files manually on the remote machine.

**Note:** *4DSLI* is also necessary to use the encryption commands **ENCRYPT BLOB** and **DECRYPT BLOB**.

The installation of these elements makes it possible to use TLS for connections to the 4D Web server. However, in order for TLS connections to be accepted by the 4D Web server, you must “activate” the TLS. This parameter is accessible on the **Configuration** tab of the **Web** page in the Database Settings:



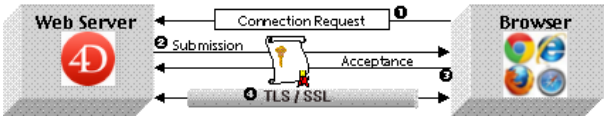

By default, the TLS connections are allowed. You can uncheck this option if you do not want to use TLS functionalities with your Web server, or if another Web server allowing secure connections is operating on the same machine.

The TCP port dedicated to TLS data exchange is 443 by default. This port number can be modified in the **HTTPS Port Number** area in order, for example, to reinforce the security of the Web server (for more information about this pont, refer to the **Web Server Settings** section). The TCP port defined on this page of the Database Settings is used for standard mode Web server connections.

**Note**: The other settings defined for the 4D Web Server management (password, timeout, cache size, etc.) are applied, regardless of whether or not the server is operating in TLS mode.

## Browser connection with TLS

For a Web connection to be carried out in secure mode, the URL sent by the browser simply needs to begin with “https” (instead of “http”).
In this case, a warning dialog appears on the browser. If the user clicks **OK**, the Web server sends the certificate to the browser.




The encryption algorithm used for the connection is then decided by the browser and the Web server. The server offers several symetric encryption algorithms (RC2, RC4, DES...). The most powerful common algorithm is used.

**Warning:** The level of encryption allowed might depend on current laws in the country of use.

### Management of the connection mode

Using TLS with a 4D Web server does not require any specific system configuration. However, you should keep in mind that a TLS Web server can also work in a non-secured mode. The connection mode can switch to another mode if the browser so requires (for example, in the browser URL area, the user can replace “HTTPS” by “HTTP”). The developer can forbid or redirect requests made in a non-secured mode. You can get the current connection mode using the **WEB Is secured connection** command.

## XML and WML Support

### WML

4D Web Server supports WML (Wireless Markup Language) technology. This feature allows a mobile phone or a PDA's owner to read and enter data in a 4D database.

**Note:** The WML language associated to the WAP (Wireless Application Protocol) is developed by several companies. The WAP technology offers a set of network communication tools so that mobile phones and PDA users can visualize text published on Web pages. The WML technology is open and free of charge. For more information on WML, please refer to the Phone.com Web site: *http://www.phone.com/*.

The data can be entered or read through WML pages using *4DTEXT* or *4DSCRIPT* tags.

Here is the list of the WML documents supported by 4D Web server:

| Extension | MIME Type | Description |
|---|---|---|
| .wml | text/vnd.wap.wml | WML pages (always supported by 4D*) |
| .wmls | text/vnd.wap.wmlscript | WML Scripts (on the client's side) |
| .wmlc | application/vnd.wap.wmlc | WML binary pages |
| .wmlsc | application/vnd.wap.wmlscript | WML binary scripts |
| .wbmp | picture/vnd.wap.wbmp | Bitmap images for mobile phones (not always supported) |

* Allows dynamic data insertion through *4DTEXT* and *4DSCRIPT* type tags.

### XML

The 4D Web server supports .xml,.xls and .dtd documents which are sent with the following MIME type: "text/xml" and "text/xsl".

4D analyzes their content and processes their *4DTEXT* or *4DSCRIPT* type tags (if any) in order to generate dynamic XML.

# WEB CLOSE SESSION

WEB CLOSE SESSION ( sessionID )

| Parameter | Type | | Description |
|---|---|---|---|
| sessionID | Text | ⇨ | Session UUID |

## Description

The **WEB CLOSE SESSION** command invalidates an existing session designated by the *sessionID* parameter. If the session does not exist, the command does nothing.

When this command is called from a Web process or any other process:

- The cookie expiration date is set to 0,
- The **On Web Close Process database method** is called, allowing you to store session information,
- Selections are erased, records are unlocked and variables are reset.

After this command is executed, if a Web client sends a request using an invalid cookie, a new session is opened and a new cookie is sent.

**Note:** In the context of a 4D Mobile session, the **WEB CLOSE SESSION** command closes the 4D Mobile session whose ID is passed in the *sessionID* parameter. Since a 4D Mobile session can manage several processes, this command actually requests all the Web processes related to the session to finish their execution.

# WEB GET BODY PART

WEB GET BODY PART ( part ; contents ; name ; mimeType ; fileName )

| Parameter | Type | | Description |
|---|---|---|---|
| part | Longint | ⇒ | Part number |
| contents | BLOB, Text | ⇐ | Contents of part |
| name | Text | ⇐ | Name of "input" variable |
| mimeType | Text | ⇐ | Mime type of submitted file |
| fileName | Text | ⇐ | Name of submitted file |

## Description

The **WEB GET BODY PART** command, when called in the context of a Web process, parses the "body" part of a multi-part request.

In the *part* parameter, pass the number of the part to be parsed. You can get the total number of parts using the **WEB Get body part count** command.

The *contents* parameter receives the contents of the part. When the parts to be retrieved are files, you must pass a BLOB type parameter. In the case of TEXT variables submitted in a Web form, you can pass a Text type parameter.

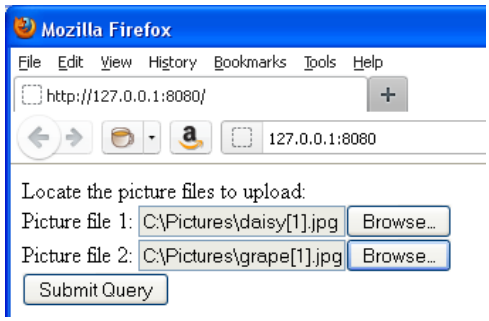
The *name* parameter receives the variable name of the HTTP input field.

The *mimeType* and *name* parameters receive the Mime type and the name of the original file, if any. A *name* is only received when the file was submitted as **<input type="file">**.
*mimeType* and *name* are optional but must be passed together.

**Note:** In the context of a multi-part request, the first array of the **WEB GET VARIABLES** command returns all parts of the form, in the same order as the **WEB GET BODY PART** command. You can use it in order to get the position of the parts of the form directly.

## Example

In this example, a Web form downloads several pictures using a browser onto the HTTP server and displays them in the page. Here is the Web form:




Here is the code for the <body> part of the page:

```
<body>        <form enctype="multipart/form-data" action="/4DACTION/GetFile/" method="post">            Locate the
picture files to upload: <br>            Picture file 1: <input name="file1" type="file"><br>            Picture file 2:
<input name="file2" type="file"><br>            <input type="submit">                                </form>            <hr/>
    <!--4DSCRIPT/galleryInit-->    <!--4Dloop aFileNames-->        <img src="/photos/<!--4Dvar aFileNames{aFileNames}--
>"/>    <!--4Dendloop--> </body>
```

Two 4D methods are called by the page:

- ***galleryInit*** on loading (4DSCRIPT tag), displays the pictures found in the destination folder.
- ***GetFile*** when sending data (4DACTION URL of multi-part form), processes the submission.

Here is the code for ***galleryInit***:

```
C_TEXT($vDestinationFolder)
ARRAY TEXT(aFileNames;0)
C_LONGINT($i)
```

```
$vDestinationFolder:=Get 4D folder(HTML Root folder)+"photos"+Folder separator //"WebFolder/photos" folder
DOCUMENT LIST($vDestinationFolder;aFileNames)
```

Here is the code for ***GetFile***:

```
C_TEXT($vPartName;$vPartMimeType;$vPartFileName;$vDestinationFolder)
C_BLOB($vPartContentBlob)
C_LONGINT($i)
$vDestinationFolder:=Get 4D folder(HTML Root folder)+"photos"+Folder separator
For($i;1;WEB Get body part count) //for each part
   WEB GET BODY PART($i;$vPartContentBlob;$vPartName;$vPartMimeType;$vPartFileName)
   If($vPartFileName#"")
      BLOB TO DOCUMENT($vDestinationFolder+$vPartFileName;$vPartContentBlob)
   End if
End for
WEB SEND HTTP REDIRECT("/") // return to page
```

# WEB Get body part count

WEB Get body part count -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Number of parts in the body |

## Description

The **WEB Get body part count** command returns the number of parts making up the body received.

## Example

Refer to the example for the **WEB GET BODY PART** command.

# WEB Get Current Session ID

WEB Get Current Session ID -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Text |  | Session UUID |

## Description

The **WEB Get Current Session ID** command returns the ID of the session open for the Web request. This ID is generated automatically by 4D as an UUID.

If this command is called outside of the context of a Web session, it returns an empty string "".

# WEB GET HTTP BODY

WEB GET HTTP BODY ( body )

| Parameter | Type | | Description |
|---|---|---|---|
| body | BLOB, Text | ⇐ | Body of the HTTP request |

## Description

The **WEB GET HTTP BODY** command returns the body of the HTTP request being processed. The HTTP body is returned as is, without processing or parsing.

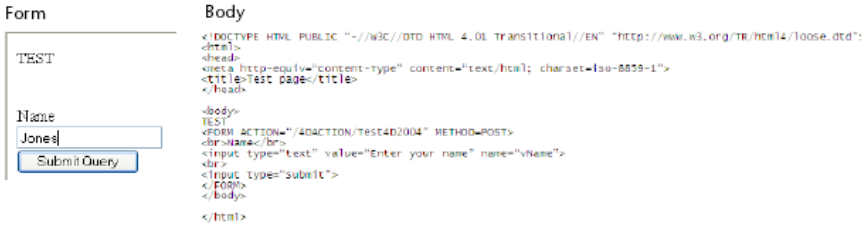
This command can be called using a Web database method (**On Web Authentication Database Method**, **On Web Connection Database Method**) or any Web method.

In *body*, you can pass a variable or a field of the BLOB or Text type. The Text type is generally sufficient (the *body* parameter can receive up to 2 GB of text).

This command allows you, for example, to carry out queries in the body of requests. It also permits advanced users to set up a WebDAV server within a 4D database.

## Example

In this example, a simple request is sent to the 4D Web server and the contents of the HTTP body are displayed in the debugger. Here is the form sent to the 4D Web server, as well as the corresponding HTML code:




Here is the Test4D2004 method:

```
C_BLOB($request)
C_TEXT($requestText)

WEB GET HTTP BODY($request)
$requestText:=BLOB to text($request;UTF8 text without length)
WEB SEND FILE("page.html")
```

**Note:** This method is declared “Available through 4D HTML tags and URLs (4DACTION...)” in its properties.

When the form is submitted to the Web server, the $requestText variable receives the text of the HTTP request body.

# WEB GET HTTP HEADER

WEB GET HTTP HEADER ( header|fieldArray {; valueArray} )

| Parameter | Type | | Description |
|---|---|---|---|
| header\|fieldArray | Text, Text array | ⇦ | Request HTTP header or HTTP header fields |
| valueArray | Text array | ⇦ | HTTP header fields content |

## Description

The **WEB GET HTTP HEADER** command returns either a string or two arrays, containing the HTTP header used for the currently processed request.

This command can be called from within any method (**On Web Authentication Database Method** or **On Web Connection Database Method**, method called by "*/4DACTION*"...) executed in a Web process.

- **First syntax: WEB GET HTTP HEADER (header)**

When this syntax is used, the result returned in the *header* variable is as follows:

```
"GET /page.html HTTP¥1.0"+Char(13)+Char(10)+"User-Agent: browser"+Char(13)+Char(10)+"Cookie: C=HELLO"
```

Each header field is separated by a CR+LF (Carriage return+Line feed) sequence under Windows and Mac OS.

- **Second syntax: WEB GET HTTP HEADER (fieldArray; valueArray)**

When this syntax is used, the returned results in the *fieldArray* and *valueArray* are as follows:

| | |
|---|---|
| *fieldArray{1} = "X-METHOD"* | *valueArray{1} = "GET" ** |
| *fieldArray{2} = "X-URL"* | *valueArray{2} = "/page.html" ** |
| *fieldArray{3} = "X-VERSION"* | *valueArray{3} = "HTTP/1.0" ** |
| *fieldArray{4} = "User-Agent"* | *valueArray{4} = "browser"* |
| *fieldArray{5} = "Cookie"* | *valueArray{5} = "C=HELLO"* |

* These first three items are not HTTP fields. They are part of the first line of the request.

To comply with the HTTP standard, field names are always written in English.

Here is a list of some HTTP fields that can be used in a request:

- **Accept**: content allowed by the browser.
- **Accept-Language**: language(s) that can be used by the browser (for information). Allows to select a web page using the language defined in the browser.
- **Cookie**: cookies list
- **From**: browser user email address.
- **Host**: server name or address (for example using an URL, http://mywebserver/mypage.html, **Host** takes the «mywebserver» value). Allows to manage several names pointing towards the same IP address (virtual hosting).
- **Referer**: request origin (for example http://mywebserver/mypage1.html), i.e. the page which is displayed when clicking on the **Previous** button.
- **User-Agent**: browser or proxy name and version.

## Example

The following method allows getting any HTTP request header field content:

```
  ` Project method GetHTTPField
  ` GetHTTPField (Text) -> Text
  ` GetHTTPField (HTTP header name) -> HTTP header content

C_TEXT($0;$1)
C_LONGINT($vlItem)
ARRAY TEXT($names;0)
```

```
ARRAY TEXT($values;0)
$0:=""
WEB GET HTTP HEADER($names;$values)
$vlItem:=Find in array($names;$1)
If($vlItem>0)
   $0:=$values{$vlItem}
End if
```

- Once this project method has been written, it can be called as follows:

```
` Cookie header content
$cookie:=GetHTTPField("Cookie")
```

- You can send different pages according to the language set in the browser (for example in the **On Web Connection Database Method**):

```
$language:=GetHTTPField("Accept-Language")
Case of
   :($language="@fr@") `French (see list ISO 639)
      WEB SEND FILE("index_fr.html")
   :($language="@sp@") `Spanish (see list ISO 639)
      WEB SEND FILE("index_es.html")
   Else
      WEB SEND FILE("index.html")
End case
```

**Note:** Web browsers allow defining several languages by default. They are listed in the "Accept-Language" field, separated by a ";". Their priority is defined according to their position within the string; therefore it is a good idea to test language positions in the string.

- Here is an example of virtual hosts (for example, in the **On Web Connection Database Method**). The following names "home_site.com", "home_site1.com" and "home_site2.com" are directed towards the same IP address, for example 192.1.2.3.

```
$host:=GetHTTPField("Host")
Case of
   :($host="www.site1.com")
      WEB SEND FILE("home_site1.com")
   :($host="www.site2.com")
      WEB SEND FILE("home_site2.com")
   Else
      WEB SEND FILE("home_site.com")
End case
```

# WEB GET OPTION

WEB GET OPTION ( selector ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| selector | Longint | ⇒ | Code of option to modify |
| value | Longint, Text | ⇐ | Value of option |

## Description

The **WEB GET OPTION** command gets the current value of an option for the 4D Web server operation.

The *selector* parameter indicates which Web option to get. In this parameter, you pass one of the constants available in the **Web Server** theme:

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Web character set | Longint | 17 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Character set that the 4D Web Server (with 4D in local mode and 4D Server) should use to communicate with browsers connecting to the database. The default value actually depends on the language of the operating system. This parameter is set in the Database settings.<br>**Possible values**: The possible values depend on the operating mode of the database relating to the character set.<br><br>• *Unicode Mode*: When the application is operating in Unicode mode, the values to pass for this parameter are character set identifiers (*MIBEnum* longint or *Name* string, identifiers defined by IANA, see the following address: http://www.iana.org/assignments/character-sets). Here is the list of identifiers corresponding to the character sets supported by the 4D Web server:<br>4=ISO-8859-1<br>12=ISO-8859-9<br>13=ISO-8859-10<br>17=Shift-JIS<br>2024=Windows-31J<br>2026=Big5<br>38=euc-kr<br>106=UTF-8<br>2250=Windows-1250<br>2251=Windows-1251<br>2253=Windows-1253<br>2255=Windows-1255<br>2256=Windows-1256<br><br>• *ASCII compatibility mode*:<br>Western European<br>1: Japanese<br>2: Chinese<br>3: Korean<br>4: User-defined<br>5: Reserved<br>6: Central European<br>7: Cyrillic<br>8: Arabic<br>9: Greek<br>10: Hebrew<br>11: Turkish<br>12: Baltic |
| Web debug log | Longint | 84 | **Scope:** Local Web server<br>**Kept between two sessions:** No, but remains valid even if the HTTP server is restarted (a new log file is used in this case)<br>**Description:** Allows you to get or set the status of the HTTP request log file of the 4D Web server. When enabled, this file, named "**HTTPDebugLog_*nn*.txt**", is stored in the "Logs" folder of the application (*nn* is the file number). It is useful for debugging issues related to the Web server. It records each request and each response in raw mode. Whole requests, including headers, are logged; optionally, body parts can be logged as well.<br>**Values**: One of the constants prefixed with "wdl" (refer to the descriptions of these constants in this theme).<br>**Default value**: 0 (not enabled) |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Web HTTP compression level | Longint | 50 | **Scope**: Local Web server<br>**Kept between two sessions**: No<br>**Description**: Compression level for all compressed HTTP exchanges for the 4D HTTP server (client requests or server replies, Web and Web Service). This selector lets you optimize exchanges by either privileging speed of execution (less compression) or the amount of compression (less speed). The choice of a value depends on the size and type of data exchanged. Pass 1 to 9 in the *value* parameter where 1 is the fastest compression and 9 the highest. You can also pass -1 to get a compromise between speed and rate of compression. By default, the compression level is 1 (faster compression).<br>**Possible values**: 1 to 9 (1 = faster, 9 = more compressed) or -1 = best compromise. |
| Web HTTP compression threshold | Longint | 51 | **Scope**: Local HTTP server<br>**Kept between two sessions**: No<br>**Description**: In the framework of optimized HTTP exchanges, size threshold for requests below which exchanges should not be compressed. This setting is useful in order to avoid losing machine time by compressing small exchanges.<br>**Possible values**: Any Longint type value. Pass the size expressed in bytes in *vaue*. By default, the compression threshold is set to 1024 bytes |
| Web HTTP TRACE | Longint | 85 | **Scope**: Local Web server<br>**Kept between two sessions**: No<br>**Description**: Allows you to disable or enable the HTTP TRACE method in the 4D Web server. For security reasons, starting with 4D v15 R2, by default the 4D Web server rejects HTTP TRACE requests with an error 405 (see HTTP TRACE disabled). If necessary, you can enable the HTTP TRACE method for the session by passing this constant with value 1. When this option is enabled, the 4D Web server replies to HTTP TRACE requests with the request line, header, and body.<br>**Possible values**: 0 (disabled) or 1 (enabled)<br>**Default value**: 0 (disabled) |
| Web HTTPS port ID | Longint | 39 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: TCP port number used by the Web server of 4D in local mode and of 4D Server for secure connections via SSL (HTTPS protocol). The HTTPS port number is set on the “Web/Configuration” page of the Database settings dialog box.<br>By default, the value is 443 (standard value). You can use the constants of the **TCP Port Numbers** theme for the *value* parameter.<br>**Possible values**: 0 to 65535 |
| Web inactive process timeout | Longint | 78 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description:** Modifies the life duration of the inactive processes associated with sessions. At the end of the timeout, the process is killed on the server, the **On Web Close Process database method** is called then the session context is destroyed.<br>**Possible values**: Longint (minutes)<br>**Default value**: 480 minutes (pass 0 to restore the default value) |
| Web inactive session timeout | Longint | 72 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description**: Modifies the life duration of inactive sessions (duration set in cookie). At the end of this period, the session cookie expires and is no longer sent by the HTTP client.<br>**Possible values**: Longint (minutes)<br>**Default value**: 480 minutes (pass 0 to restore the default value) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Web IP address to listen | Longint | 16 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: IP address on which the 4D Web server will receive HTTP requests with 4D in local mode and 4D Server. By default, no specific address is defined (*value* = 0). This parameter can be set in the Database settings. The *Web IP Address to listen* selector is useful for 4D Web Servers compiled and merged with 4D Desktop (in which there is no access to the Design mode).<br>You pass a hexadecimal IP address in the *value* parameter. In other words, to designate a IP address such as "a.b.c.d", you should write:<br>`C_LONGINT($addr)`<br>`$addr:=($a<<24)\|($b<<16)\|($c<<8)\|$d`<br>`WEB SET OPTION(Web IP address to listen;$addr)` |
| Web keep session | Longint | 70 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description:** Enables or disables the session management mode (described in the **Web Sessions Management** section)<br>**Possible values**: 1 (enable mode) or 0 (disable mode)<br>**Default value**: 1 for databases created in v13, 0 for converted databases. Note that this mode also enables the mechanism for reusing temporary contexts in remote mode. For more information about this mechanism, refer to the description of this option in the **Web Server Settings** section. |
| Web log recording | Longint | 29 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Starts or stops the recording of Web requests received by the Web server of 4D in local mode or 4D Server. By default, the value is 0 (requests not recorded).<br>The log of Web requests is stored as a text file named "logweb.txt" that is automatically placed in the Logs folder of the database, next to the structure file. The format of this file is determined by the value that you pass. For more information about Web log file formats, please refer to the **Information about the Web Site** section.<br>This file can also be activated on the "Web/Log" page of the Database settings.<br>**Possible values**: 0 = Do not record (default), 1 = Record in CLF format, 2 = Record in DLF format, 3 = Record in ELF format, 4 = Record in WLF format.<br>**Warning**: Formats 3 and 4 are custom formats whose contents must be set beforehand in the Database settings. If you use one of these formats without any of its fields having been selected on this page, the log file will not be generated. |
| Web max concurrent processes | Longint | 18 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Strictly upper limit of concurrent Web processes of any type supported by the 4D Web Server with 4D in local mode and 4D Server. When this number (minus one) is reached, 4D will not create any other processes and returns the HTTP status 503 - Service Unavailable to all new requests.<br>This parameter can prevent the 4D Web Server from saturation, which can occur when an exceedingly large number of concurrent requests are sent, or when too many context creations are requested. This parameter can also be set in the Database settings.<br>In theory, the maximum number of Web processes is the result of the following formula: Available memory/Web process stack size. Another solution is to view the information on Web processes displayed in the Runtime Explorer: the current number of Web processes and the maximum number reached since the Web server boot are indicated.<br>**Possible values**: Any value between 10 and 32 000. The default value is 100. |
| Web max sessions | Longint | 71 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description**: Limits the number of simultaneous sessions. When you reach the limit set, the oldest session is closed (and **On Web Close Process database method** is called) if the Web server needs to create a new one.<br>**Possible values**: Longint. The number of simultaneous sessions cannot exceed the total number of Web processes (Web Max Concurrent Processes option, 100 by default)<br>**Default value**: 100 (pass 0 to restore the default value) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Web maximum requests size | Longint | 27 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Maximum size (in bytes) of incoming HTTP requests (POST) that the Web server is authorized to process. By default, the value is 2 000 000, i.e. a little less than 2 MB. Passing the maximum value (2 147 483 648) means that, in practice, no limit is set. This limit is used to avoid Web server saturation due to incoming requests that are too large. When a request reaches this limit, the 4D Web server refuses it.<br>**Possible values**: 500 000 to 2 147 483 648. |
| Web port ID | Longint | 15 | **Scope**: 4D in local mode and 4D Server.<br>**Kept between two sessions:** No<br>**Description:** Sets or gets the number of the TCP port used by the 4D Web server with 4D in local mode and 4D Server. By default, the value is 80. The TCP port number is set on the "Web/Configuration" page of the Database Settings dialog box. You can use one of the constants in the **TCP Port Numbers** theme for the *value* parameter. This selector is useful within the framework of 4D Web servers that are compiled and merged using 4D Desktop (no access to the Design environment).<br>**Possible values:** For more information about the TCP port number, refer to the **Web Server Settings** section.<br>**Default value:** 80 |
| Web session cookie domain | Longint | 81 | **Scope:** local Web server<br>**Kept between two sessions:** No, but remains valid even if the HTTP server is restarted.<br>**Description:** Sets or gets the value of the "domain" field of the session cookie. This selector (as well as selector 82) is useful for controlling the scope of the session cookies: If you set, for example, the value "/*.4d.fr" for this selector, the client will only send a cookie when the request is addressed to the domain ".4d.fr", which excludes servers hosting external static data.<br>**Possible values:** Text |
| Web session cookie name | Longint | 73 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description:** Sets the name of the cookie used for saving the session ID.<br>**Possible values:** Text<br>**Default value:** "4DSID" (pass an empty string to restore the default value) |
| Web session cookie path | Longint | 82 | **Scope:** local Web server<br>**Kept between two sessions:** No, but remains valid even if the HTTP server is restarted.<br>**Description:** Sets or gets the value of the "path" field of the session cookie. This selector (as well as selector 81) is useful for controlling the scope of the session cookies: If you set, for example, the value "/4DACTION" for this selector, the client will only send a cookie for dynamic requests beginning with 4DACTION, and not for pictures, static pages, etc.<br>**Possible values:** Text |
| Web session enable IP address validation | Longint | 83 | **Scope**: Local Web server<br>**Kept between two sessions**: No<br>**Description**: Enables or disables IP address validation for session cookies. For security reasons, by default the 4D Web server checks the IP address of each request containing a session cookie and rejects it if this address does not match the IP address used to create the cookie. In some specific applications, you may want to disable this validation and accept session cookies, even when their IP addresses do not match. For example when mobile devices switch between Wifi and 3G/4G networks, their IP address will change. In this case, you must pass 0 in this option to allow clients to be able to continue using their Web sessions even when the IP addresses change. Note that this setting lowers the security level of your application.<br>When it is modified, this setting is effective immediately (you do not need to restart the HTTP server).<br>**Possible values**: 0 (disabled) or 1 (enabled)<br>**Default value**: 1 (IP addresses are checked) |

When you use the Web debug log *selector*, you can receive one of the following constants in the *value* parameter:

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| wdl disable | Longint | 0 | Web HTTP debug log is disabled |
| wdl enable with all body parts | Longint | 7 | Web HTTP debug log is enabled with body parts in response and request |
| wdl enable with request body | Longint | 5 | Web HTTP debug log is enabled with body part in request only |
| wdl enable with response body | Longint | 3 | Web HTTP debug log is enabled with body part in response only |
| wdl enable without body | Longint | 1 | Web HTTP debug log is enabled without body parts (body size is provided in this case) |

# WEB GET SESSION EXPIRATION

WEB GET SESSION EXPIRATION ( sessionID ; expDate ; expTime )

| Parameter | Type | | Description |
|---|---|---|---|
| sessionID | Text | ⇨ | Session UUID |
| expDate | Date | ⇦ | Date of cookie expiration |
| expTime | Time | ⇦ | Time of cookie expiration |

## Description

The **WEB GET SESSION EXPIRATION** command returns the expiration information for the cookie of the session whose UUID you passed in *sessionID*.

The *expDate* parameter receives the expiration date and the *expTime* parameter receives the expiration time of the cookie.

**Note:** Each time a web request is sent, the expiration date and time of the cookie are reset to a value corresponding to the time of the request+the value of the Web Inactive session timeout. For example:
*First request, Monday at 1:00*
-> Sends a cookie 4DSID xxxyyy expiration I+24h = Tuesday 01:00
*Second request, Monday at 1:10*
-> Sends a cookie 4DSID xxxyyy expiration I+24h = Tuesday 01:10
*Third request, Tuesday at 4:00: cookie expired*
-> Sends a cookie 4DSID aaabbb expiration I+24h = Wednesday 01:00

# WEB Get session process count

WEB Get session process count ( sessionID ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| sessionID | Text | ⇒ | Session UUID |
| Function result | Longint | ⇐ | Number of processes attached to the session |

## Description

The **WEB Get session process count** command returns the number of running processes attached to the session whose UUID you passed in *sessionID*.

This command is added in the context of the **Handling of 4D Mobile sessions by programming** feature introduced in 4D v15 R4. It is mainly designed to count the number of processes run by a 4D Mobile session.

- For a 4D Mobile session, this command returns the actual number of processes. A 4D Mobile session can run several processes.
- For a regular Web session, this command always return 1 (one Web session = one process).

## Example

You want to store information on the current 4D Mobile session in arrays:

```
C_TEXT($sessionID)
C_LONGINT($count)
C_DATE($expDate)
C_TIME($expTime)

$sessionID:=WEB Get Current Session ID
$count:=WEB Get session process count($sessionID)
WEB GET SESSION EXPIRATION($sessionID;$expDate;$expTime)

APPEND TO ARRAY($aTimestamp;String(Current date)+" "+String(Current time))
APPEND TO ARRAY($aSessionUID;$sessionID)
APPEND TO ARRAY($aNbProcesses;$count)
APPEND TO ARRAY($aExpirationDate;$expDate)
APPEND TO ARRAY($aExpirationTime;$expTime)
```

# WEB GET STATISTICS

WEB GET STATISTICS ( pages ; hits ; usage )

| Parameter | Type | | Description |
|---|---|---|---|
| pages | Text array | ⇦ | Names of the most consulted pages |
| hits | Longint array | ⇦ | Number of hits for each page |
| usage | Longint | ⇦ | Percentage of the cache used |

## Description

The **WEB GET STATISTICS** command lets you get information about the most consulted pages loaded in the Web server's cache. Consequently, these statistics only concern static pages, GIF pictures, JPEG pictures <100 KB and style sheets (.css).

**Note:** For more information about setting the 4D Web server's cache, refer to the **Web Server Settings** section.

The command fills the pages Text array with the names of the most consulted pages. The *hits* Longint array receives the number of "hits" for each page. The *usage* parameter receives the percentage of the Web cache used by each page.

## Example

Let's assume that you want to generate a semi-dynamic page that displays the statistics of the Web cache. For this, in a static HTML page named "stats.shtm" (pages suffixed .shtm are automatically parsed by the Web server), you place the tag *<!--#4DSCRIPT/STATS-->* as well as references to the *vPages* and *vUsage* variables, for example:

```
<html> <head><title>4D Web Stats</title></head> <!--#4DSCRIPT/STATS--> <body> <strong>Percentage of cache used: </strong>
<!--#4DTEXT vUsage--> <hr> <strong>Pages consulted most often: </strong> <!--#4DHTML vPages--> </body> </html>
```

In the project method ***STATS***, you write the following code:

```
C_TEXT($1)
C_TEXT(vPages)
ARRAY TEXT(pages;0)
ARRAY LONGINT(hits;0)
C_LONGINT(vUsage)

WEB CACHE STATISTICS(pages;hits;vUsage)
For($i;1;Size of array(pages))
   ¥¥For each page present in the cache
   vPages:=pages{$i}+"   "+String(hits{$i})+"<br>"
   ¥¥Insert the name of the page and the HTML code
End for
```

You can send the "stats.shtm" page using a URL link or using the **WEB SEND FILE** command.

# WEB GET VARIABLES

WEB GET VARIABLES ( nameArray ; valueArray )

| Parameter | Type | | Description |
|---|---|---|---|
| nameArray | Text array | ⇐ | Web form variable names |
| valueArray | Text array | ⇐ | Web form variable values |

## Description

The **WEB GET VARIABLES** command fills the text arrays *nameArray* and *valueArray* with the variable names and values contained in the Web form "submitted" (i.e. sent to the Web server).

This command gets the value for all the variables which can be included in HTML pages: text area, button, check box, radio button, pop up menu, choice list...

**Note:** Regarding check boxes, the variable name and value are returned only if the check box has been actually checked.

This command is valid regardless of the type of URL or form (POST or GET method) sent to the Web server.

This command can be called, if necessary, in the **On Web Connection Database Method** or any other 4D method resulting from a form submission.

### About Web forms and their associated actions

Each form contains named data entry area (text area, buttons, checkboxes).
When a form is submitted (a request is sent to the Web server), the request contains (within others) the list of the data entry areas and their associated values.
A form can be submitted through two methods (both can be used with 4D):

- POST, usually used to add data into the Web server - to a database,
- GET, usually used to request the Web server - data coming from a database.

## Example

A form contains two fields, vName and vCity with "ROBERT" and "DALLAS" values. The action associated to the form is "/4DACTION/WEBFORM".

- If the form method is POST (most frequently used), the data entered will not be visible in the URL (http://127.0.0.1/4DACTION/WEBFORM).
- If the form method is GET, the data entered will be visible in the URL (http://127.0.0.1/4DACTION/WEBFORM?vNAME=ROBERT&vCITY=DALLAS).

The WEBFORM method can be as follows:

```
ARRAY TEXT($anames;0)
ARRAY TEXT($avalues;0)
WEB GET VARIABLES($anames;$avalues)
```

The result will be:

```
$anames{1}="vNAME"
$anames{2}="vCITY"
$avalues{1}="ROBERT"
$avalues{2}="DALLAS"
```

The vNAME variable contains ROBERT and the vCITY variable contains DALLAS.

# WEB Is secured connection

WEB Is secured connection -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ➲ | True = the web connection is secured. False = the web connection is not secured. |

## Description

The **WEB Is secured connection** command returns a Boolean indicating if the 4D Web server connection was done in secured mode through TLS/SSL (the request starts with “https:” instead of “http:”).

- If the connection is made through TLS or SSL, the function returns True.
- If the connection is made in a non-secured mode, the function returns False.

**Note:** For more information on the TLS protocol, refer to the **Using TLS Protocol** section.

This command allows, for example, denying connections made in a non-secured mode (if any).

# WEB Is server running

WEB Is server running -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | True if the Web Server is running, otherwise False |

## Description

The new **WEB Is server running** command returns **True** if the 4D built-in Web server is running, and **False** if the Web server is off.

This command returns the running status of the Web Server where it is executed:

| Execution context | Returns the status of |
|---|---|
| 4D stand-alone application | Local 4D Web server |
| 4D Server | 4D Server Web server |
| 4D remote mode (local process) | Local 4D Web server |
| 4D remote mode (4D Server stored procedure) | 4D Server Web server |
| 4D remote mode (other 4D remote stored procedure) | Remote 4D Web server |

## Example

You want to check that the Web server is running:

```
If(WEB Is server running)
   ... //do appropriate actions
End if
```

# WEB SEND BLOB

WEB SEND BLOB ( blob ; type )

| Parameter | Type | | Description |
|---|---|---|---|
| blob | BLOB | ⇒ | BLOB to send to the browser |
| type | String | ⇒ | Data type of the BLOB |

## Description

The **WEB SEND BLOB** command allows you to send *blob* to the browser.

The type of data contained in the BLOB is indicated by *type*. This parameter can be one of the following types:

- *type* = **Empty String** (""): In this case, you don't need to supply any more information in the BLOB. The browser will try to interpret the contents of the BLOB.
- *type* = **File extension** (example: ".HTM", ".GIF", ".JPEG", etc.): In this case, you specify the MIME type of the data contained in the BLOB by indicating its extension. The BLOB will then be interpreted according to its extension. However, the extension must be a standard one so that the browser can correctly interpret it.
- *type* = **Mime/Type** (example: "text/html", "image/tiff", etc.): In this case, you directly specify the MIME type of data contained in the BLOB. This solution offers you more freedom. Besides the standard types, you can pass a custom MIME type to send proprietary documents via Intranet. To do so, you only need to configure the browsers so that they recognize the type sent and so that they can open the appropriate application. The value you pass to *type* is, in this case, "application/x-[TypeName]". In the client workstations's browser, you reference this type and associate it to the "Launch the application" action. The **WEB SEND BLOB** command allows you to therefore send all types of documents, the Intranet clients automatically open the associated application.

  **Note:** For more information about MIME types, refer to the page: http://www.iana.org/assignments/media-types.

Here is a list of the most common MIME types:

| Extension | Mime/Type |
|---|---|
| .htm | text/html |
| .html | text/html |
| .shtml | text/html |
| .shtm | text/html |
| .css | text/css |
| .pdf | application/pdf |
| .rtf | application/rtf |
| .ps | application/postscript |
| .eps | application/postscript |
| .hqx | application/mac-binhex40 |
| .js | application/javascript |
| .json | application/json |
| .txt | text/plain |
| .text | text/plain |
| .gif | image/gif |
| .jpg | image/jpeg |
| .jpeg | image/jpeg |
| .jpe | image/jpeg |
| .jfif | image/jpeg |
| .pic | image/pict |
| .pict | image/pict |
| .tif | image/tiff |
| .tiff | image/tiff |
| .mpeg | video/mpeg |
| .mpg | video/mpeg |
| .mov | video/quicktime |
| .moov | video/quicktime |
| .aif | audio/aiff |
| .aiff | audio/aiff |
| .wav | audio/wav |
| .ram | audio/x-pn-realaudio |
| .sit | application/x-stuffit |
| .bin | application/x-stuffit |
| .xml | application/xml |
| .z | application/x-zip |
| .zip | application/x-zip |
| .gz | application/x-gzip |
| .tar | application/x-tar |

**Note:** The list of MIME types supported by the 4D HTTP server is saved in the "MimeTypes.xml" file found in the following folder of the 4D application: *[Contents]¥Native components¥HTTPServer.bundle¥Contents¥Resources*.

The references to 4D variables and *4DSCRIPT* type tags in the page are always parsed.

## Example

Refer to the example of the **PICTURE TO GIF** routine.

# WEB SEND FILE

WEB SEND FILE ( htmlFile )

| Parameter | Type | | Description |
|---|---|---|---|
| htmlFile | String | ⇒ | HTML Pathname to HTML file or empty string for terminating SEND HTML FILE |

## Description

The **WEB SEND FILE** command sends, to the Web browser, the HTML page or the Web file stored in the document whose pathname you pass in *htmlFile*.

By default, 4D looks for the HTML document within the root folder, defined in the Database Settings.

This command accepts as a parameter either pathnames in Posix syntax (names of directories or folders are separated with a slash "/") or in the system syntax.
Specifying an invalid pathname generates an error related to file management for your operating system. You can intercept this error using a method installed by the **ON ERR CALL** command. If the method displays a warning or message dialog box, it will appear on the browser machine.

Once **WEB SEND FILE** is executed, the OK system variable is updated: if the file to be sent exists and if the timeout has not run out, OK is equal to 1. Otherwise, it is equal to 0.

**Note:** If you call **WEB SEND FILE** from within a process that is not a Web process, the command does nothing and returns no error; the call is simply ignored.

The references to 4D variables and *4DSCRIPT* type tags found on the page are parsed when the document type allows for it (document based on text).

## Example

The HTML root folder of the database is the *WebDocs* folder. It contains the following elements:

```
..¥WebDocs¥HTM¥MyPage.HTM
```

Sending the Web page *"MyPage.HTM"* must be carried out in the following manner :

```
WEB SEND FILE("HTM/MyPage.HTM")
```

## System variables and sets

If the file to be sent exists and if the timeout has not run out, OK is set to 1. Otherwise, it is equal to 0.

# WEB SEND HTTP REDIRECT

WEB SEND HTTP REDIRECT ( url {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| url | String | ⇒ | New URL |
| * | Operator | ⇒ | If specified = URL is not translated, If omitted = URL is translated |

## Description

The **WEB SEND HTTP REDIRECT** command allows you to transform a URL into another one.

The *url* parameter contains the new URL that allows you to redirect the request. If this parameter is a url to a file, it must contain the reference to this file, for example: **WEB SEND HTTP REDIRECT** ("/MyPage.HTM").

This command prevails over commands that send data (**WEB SEND FILE**, **WEB SEND BLOB**, etc.) that may be in the same method.

This command also allows you to redirect a request to another Web server.

4D automatically encodes the URL's special characters. If you pass the * character, 4D will not translate them.

Note that the status of the request sent by this command is **302: Moved Temporarily**. If you need a "moved permanently" status (status 301), you can set a HTTP *X-STATUS: 301* field in the header of the reply.

## Example

You can use this command to execute custom requests in 4D by using static pages. Imagine that you have placed the following elements in a static HTML page:


**Note:** The POST action "/4dcgi/rech" has been associated to the text area and to the **OK** and **Cancel** buttons.

In the **On Web Connection Database Method**, you insert the following code:

```
Case of
   :($1="/4dcgi/rech") //When 4D receives this URL
 //If the OK button has been used and the 'name' field contains a Value
      If((bOK="OK") & (name#""))
 //Change the URL to execute the request code,
 //placed farther down in the same method
         WEB SEND HTTP REDIRECT("/4dcgi/rech?"+name)
      Else
 //Else return to the beginning page
         WEB SEND HTTP REDIRECT("/page1.htm")
      End if
      ...
   :($1="/4dcgi/rech?@") //If the URL has been redirected
      ... //Put the request code here
End case
```

# WEB SEND RAW DATA

WEB SEND RAW DATA ( data {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| data | BLOB | ➡ | HTTP data to send |
| * | Operator | ➡ | Send chunked |

## Description

The **WEB SEND RAW DATA** command lets the 4D Web server send “raw” HTTP data, which can be chunked. .

The *data* parameter contains the two standard parts of an HTTP response, i.e. Header and Body. The data are sent without prior formatting by the server. However, 4D carries out a basic check of the response header and body in order to make sure that they are valid:

- If the header is incomplete or does not comply with the HTTP protocol specifications, 4D will change it accordingly.
- If the HTTP request is incomplete, 4D adds the missing information. If, for instance, you want to redirect the request, you must write:

```
HTTP/1.1 302    Location: http://...
```

If you only pass:

```
Location: http://...
```

4D will complete the request by adding *HTTP/1.1 302*.

The optional * parameter lets you specify that the response will be sent “chunked”. The cutting up of responses into chunks can be useful when the server sends a response without knowing its total length (if, for instance, the response has not yet been generated). All HTTP/1.1-compatible browsers accept chunked responses.

If you pass the * parameter, the Web server will automatically include the *transfer-encoding: chunked* field in the header of the response, if necessary (you can handle the response header manually if you so desire).

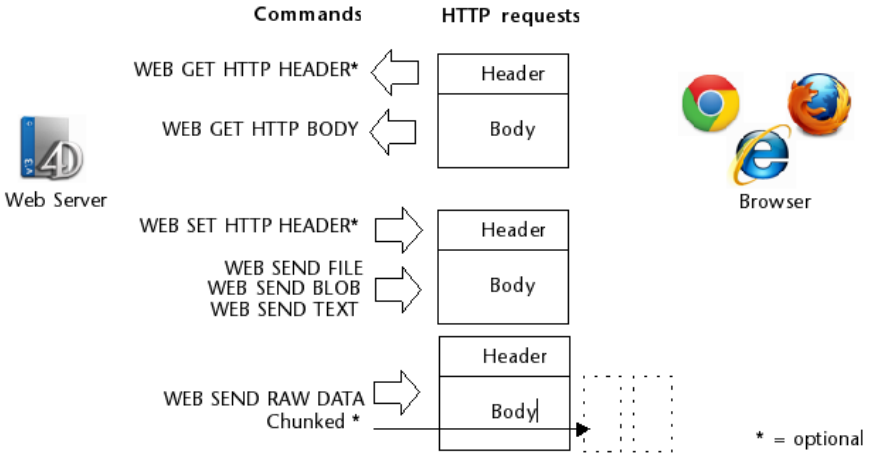
The remainder of the response will also be formatted in order to respect the syntax of the chunked option. Chunked responses contain a single header and an undefined number of body “chunks”.

All the **WEB SEND RAW DATA** statements that follow the execution of **WEB SEND RAW DATA**(data;*) within the same method will be considered as part of the response (regardless of whether they contain the * parameter). The server puts an end to the chunked send when the method execution is terminated.

**Note:** If the Web client does not support HTTP/1.1, 4D will automatically convert the response into an HTTP/1.0-compatible format (the data sent will not be chunked). However, in this case, the result may not correspond to your wishes. It is therefore recommended to check whether the Web browser supports HTTP/1.1 and to send an appropriate response. To do so, you can use a method such as:

```
C_BOOLEAN($0)
ARRAY TEXT(arFields;0)
ARRAY TEXT(arValues;0)
WEB GET HTTP HEADER(arFields;arValues)
$0:=False
If(Size of array(arValues)>=3)
   If(Position("HTTP/1.1";arValues{3})>0)
      $0:=True //The browser supports HTTP/1.1; $0 returns True
   End if
End if
```

Combined with the **WEB GET HTTP BODY** command and other commands of the “Web Server” theme, this command completes the range of tools available to 4D developers in order to entirely customize the processing of incoming and outgoing HTTP connections. These different tools are shown in the following diagram:




## Example

This example illustrates the use of the chunked option with the **WEB SEND RAW DATA** command. The data (a sequence of numbers) are sent in 100 chunks generated on the fly in a loop. Keep in mind that the header of the response is not explicitly set: the command will send it automatically and insert the *transfer-encoding: chunked* field into it since the * parameter is used.

```
C_LONGINT($cpt)
C_BLOB($my_blob)
C_TEXT($output)

For($cpt;1;100)
   $output:="["+String($cpt)+"]"
   TEXT TO BLOB($output;$my_blob;UTF8 text without length)
   WEB SEND RAW DATA($my_blob;*)
End for
```

# WEB SEND TEXT

WEB SEND TEXT ( htmlText {; type} )

| Parameter | Type | | Description |
|---|---|---|---|
| htmlText | Text | ⇒ | HTML text field or variable to be sent to the Web browser |
| type | Text | ⇒ | MIME type |

## Description

The **WEB SEND TEXT** command directly sends HTML formatted text data.

The *htmlText* parameter contains the data to be sent. Since 4D does not check the parameter content, make sure that the HTML encoding is correct.

Any references to 4D variables and *4DSCRIPT* type tags in the text are always parsed.

By default, if you omit the *type* parameter, 4D assumes that the data sent is of the "text/html" type. The command is then exactly the same as sending a BLOB of the "text/html" type using the **WEB SEND BLOB** command.
You can also use the *type* parameter to specify the MIME type of the text to be sent. For more information about the MIME types supported, refer to the description of the **WEB SEND BLOB** command.

## Example

The following method:

```
TEXT TO BLOB("<html><head></head><body>"+String(Current time)+"</body></html>";$blob;UTF8 Text without length)
WEB SEND BLOB($blob;"text/html")
```

... can be replaced by the single line:

```
WEB SEND TEXT("<html><head></head><body>"+String(Current time)+"</body></html>")
```

# WEB SET HOME PAGE

WEB SET HOME PAGE ( homePage )

| Parameter | Type | | Description |
|---|---|---|---|
| homePage | String | ⇒ | Page name or HTML access path to the page or "" to not send the custom home page |

## Description

The **WEB SET HOME PAGE** command allows you to modify the custom home page for the current Web process.

The defined page is linked to the Web process, you can therefore define the different home pages depending, for example, on the user that is connected. This page can either be static or semi-dynamic.

You pass the name of the HTML home page or the page's HTML access path to the *homePage* parameter.

**Note:** If the page specified in the *homePage* parameter does not exist when the Web process accesses it for the first time, the Web server creates it and assigns it the contents of the default home page (see **Defining a Default Home Page**).

To stop sending *homePage* as home page for the current Web process, execute **WEB SET HOME PAGE** with an empty string ("") passed in *homePage*.

**Note:** The default home page of the Web server is specified in the Database Settings dialog box.

# WEB SET HTTP HEADER

WEB SET HTTP HEADER ( header|fieldArray {; valueArray} )

| Parameter | Type | | Description |
|---|---|---|---|
| header\|fieldArray | Text, Text array | ⇒ | Field or variable containing the request HTTP header or HTTP header fields |
| valueArray | Text array | ⇒ | HTTP header field content |

## Description

The **WEB SET HTTP HEADER** command allows you to set the fields in the HTTP header of the reply sent to the Web browser by 4D. It only has an effect in a Web process.
This command allows you to manage “cookies”.

Two syntaxes are available for this command:

- First syntax: **WEB SET HTTP HEADER** (header)
  You pass the HTTP header fields to the *fields* parameter, of the Text type (variable or field), that you want to set. This syntax allows writing header types such as "*HTTP/1.0 200 OK"+Char(13)+"Set-Cookie: C=HELLO*". Header fields must be separated by the CR or CR+LF (Carriage return + Line feed) sequence, under Windows and Mac OS, 4D formats the reply.

Here is an example of a custom “cookie”:

```
C_TEXT($vTcookie)
$vTcookie:="Set-Cookie: USER="+String(Abs(Random))+"; PATH=/"
WEB SET HTTP HEADER($vTcookie)
```

**Note:** The command will not accept a literal text type constant as the *header* parameter; it must be a 4D variable or field.

For more information about the syntax, please refer to the R.F.Cs (Request For Comments) that can be found at the following Internet address: http://www.w3c.org.

- Second syntax: **WEB SET HTTP HEADER** (fieldArray; valueArray)
  The HTTP header is defined through two text arrays, *fieldArray* and *valueArray*. The header will be written as follows:

```
fieldArray{1}:="X-VERSION"
fieldArray{2}:="X-STATUS"
fieldArray{3}:="Set-Cookie"
fieldArray{4}:="Server"

valueArray{1}:="HTTP/1.0"*
valueArray{2}:="200 OK"*
valueArray{3}:="C=HELLO"
valueArray{4}:="North_Carolina"
```

* The first two items are the first line of the reply. When they are entered, they must be the first and second items of the arrays. However, it is possible to omit them and to write only the following (4D will handle the header format):

```
fieldArray{1}:="Set-Cookie"
valueArray{1}:="C=HELLO"
```

If you do not specify a state, it will automatically be HTTP/1.0 200 OK. By default, the **Server** field is "4D/<version>". The **Date** and **Content-Length** fields are also set by default by 4D.

# WEB SET OPTION

WEB SET OPTION ( selector ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| selector | Longint | ⇒ | Option code |
| value | Longint, Text | ⇒ | Option value |

## Description

The **WEB SET OPTION** command modifies the current value of various options concerning the functioning of the 4D Web server.

In the *selector* parameter, pass one of the constants from the **Web Server** theme and pass the new value of the option in *value*:

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Web character set | Longint | 17 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Character set that the 4D Web Server (with 4D in local mode and 4D Server) should use to communicate with browsers connecting to the database. The default value actually depends on the language of the operating system. This parameter is set in the Database settings.<br>**Possible values**: The possible values depend on the operating mode of the database relating to the character set.<br>• *Unicode Mode*: When the application is operating in Unicode mode, the values to pass for this parameter are character set identifiers (*MIBEnum* longint or *Name* string, identifiers defined by IANA, see the following address: http://www.iana.org/assignments/character-sets). Here is the list of identifiers corresponding to the character sets supported by the 4D Web server:<br>4=ISO-8859-1<br>12=ISO-8859-9<br>13=ISO-8859-10<br>17=Shift-JIS<br>2024=Windows-31J<br>2026=Big5<br>38=euc-kr<br>106=UTF-8<br>2250=Windows-1250<br>2251=Windows-1251<br>2253=Windows-1253<br>2255=Windows-1255<br>2256=Windows-1256<br>• *ASCII compatibility mode*:<br>Western European<br>1: Japanese<br>2: Chinese<br>3: Korean<br>4: User-defined<br>5: Reserved<br>6: Central European<br>7: Cyrillic<br>8: Arabic<br>9: Greek<br>10: Hebrew<br>11: Turkish<br>12: Baltic |
| Web debug log | Longint | 84 | **Scope:** Local Web server<br>**Kept between two sessions:** No, but remains valid even if the HTTP server is restarted (a new log file is used in this case)<br>**Description:** Allows you to get or set the status of the HTTP request log file of the 4D Web server. When enabled, this file, named "**HTTPDebugLog_*nn*.txt**", is stored in the "Logs" folder of the application (*nn* is the file number). It is useful for debugging issues related to the Web server. It records each request and each response in raw mode. Whole requests, including headers, are logged; optionally, body parts can be logged as well.<br>**Values**: One of the constants prefixed with "wdl" (refer to the descriptions of these constants in this theme).<br>**Default value**: 0 (not enabled) |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Web HTTP compression level | Longint | 50 | **Scope**: Local Web server<br>**Kept between two sessions**: No<br>**Description**: Compression level for all compressed HTTP exchanges for the 4D HTTP server (client requests or server replies, Web and Web Service). This selector lets you optimize exchanges by either privileging speed of execution (less compression) or the amount of compression (less speed). The choice of a value depends on the size and type of data exchanged. Pass 1 to 9 in the *value* parameter where 1 is the fastest compression and 9 the highest. You can also pass -1 to get a compromise between speed and rate of compression. By default, the compression level is 1 (faster compression).<br>**Possible values**: 1 to 9 (1 = faster, 9 = more compressed) or -1 = best compromise. |
| Web HTTP compression threshold | Longint | 51 | **Scope**: Local HTTP server<br>**Kept between two sessions**: No<br>**Description**: In the framework of optimized HTTP exchanges, size threshold for requests below which exchanges should not be compressed. This setting is useful in order to avoid losing machine time by compressing small exchanges.<br>**Possible values**: Any Longint type value. Pass the size expressed in bytes in *vaue*. By default, the compression threshold is set to 1024 bytes |
| Web HTTP TRACE | Longint | 85 | **Scope**: Local Web server<br>**Kept between two sessions**: No<br>**Description**: Allows you to disable or enable the HTTP TRACE method in the 4D Web server. For security reasons, starting with 4D v15 R2, by default the 4D Web server rejects HTTP TRACE requests with an error 405 (see HTTP TRACE disabled). If necessary, you can enable the HTTP TRACE method for the session by passing this constant with value 1. When this option is enabled, the 4D Web server replies to HTTP TRACE requests with the request line, header, and body.<br>**Possible values**: 0 (disabled) or 1 (enabled)<br>**Default value**: 0 (disabled) |
| Web HTTPS port ID | Longint | 39 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: TCP port number used by the Web server of 4D in local mode and of 4D Server for secure connections via SSL (HTTPS protocol). The HTTPS port number is set on the “Web/Configuration” page of the Database settings dialog box.<br>By default, the value is 443 (standard value). You can use the constants of the **TCP Port Numbers** theme for the *value* parameter.<br>**Possible values**: 0 to 65535 |
| Web inactive process timeout | Longint | 78 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description:** Modifies the life duration of the inactive processes associated with sessions. At the end of the timeout, the process is killed on the server, the **On Web Close Process database method** is called then the session context is destroyed.<br>**Possible values**: Longint (minutes)<br>**Default value**: 480 minutes (pass 0 to restore the default value) |
| Web inactive session timeout | Longint | 72 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description**: Modifies the life duration of inactive sessions (duration set in cookie). At the end of this period, the session cookie expires and is no longer sent by the HTTP client.<br>**Possible values**: Longint (minutes)<br>**Default value**: 480 minutes (pass 0 to restore the default value) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Web IP address to listen | Longint | 16 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: IP address on which the 4D Web server will receive HTTP requests with 4D in local mode and 4D Server. By default, no specific address is defined (*value* = 0). This parameter can be set in the Database settings. The *Web IP Address to listen* selector is useful for 4D Web Servers compiled and merged with 4D Desktop (in which there is no access to the Design mode).<br>You pass a hexadecimal IP address in the *value* parameter. In other words, to designate a IP address such as "a.b.c.d", you should write:<br>`C_LONGINT($addr)`<br>`$addr:=($a<<24)\|($b<<16)\|($c<<8)\|$d`<br>`WEB SET OPTION(Web IP address to listen;$addr)` |
| Web keep session | Longint | 70 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description:** Enables or disables the session management mode (described in the **Web Sessions Management** section)<br>**Possible values**: 1 (enable mode) or 0 (disable mode)<br>**Default value**: 1 for databases created in v13, 0 for converted databases. Note that this mode also enables the mechanism for reusing temporary contexts in remote mode. For more information about this mechanism, refer to the description of this option in the **Web Server Settings** section. |
| Web log recording | Longint | 29 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Starts or stops the recording of Web requests received by the Web server of 4D in local mode or 4D Server. By default, the value is 0 (requests not recorded).<br>The log of Web requests is stored as a text file named "logweb.txt" that is automatically placed in the Logs folder of the database, next to the structure file. The format of this file is determined by the value that you pass. For more information about Web log file formats, please refer to the **Information about the Web Site** section.<br>This file can also be activated on the "Web/Log" page of the Database settings.<br>**Possible values**: 0 = Do not record (default), 1 = Record in CLF format, 2 = Record in DLF format, 3 = Record in ELF format, 4 = Record in WLF format.<br>**Warning**: Formats 3 and 4 are custom formats whose contents must be set beforehand in the Database settings. If you use one of these formats without any of its fields having been selected on this page, the log file will not be generated. |
| Web max concurrent processes | Longint | 18 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Strictly upper limit of concurrent Web processes of any type supported by the 4D Web Server with 4D in local mode and 4D Server. When this number (minus one) is reached, 4D will not create any other processes and returns the HTTP status 503 - Service Unavailable to all new requests.<br>This parameter can prevent the 4D Web Server from saturation, which can occur when an exceedingly large number of concurrent requests are sent, or when too many context creations are requested. This parameter can also be set in the Database settings.<br>In theory, the maximum number of Web processes is the result of the following formula: Available memory/Web process stack size. Another solution is to view the information on Web processes displayed in the Runtime Explorer: the current number of Web processes and the maximum number reached since the Web server boot are indicated.<br>**Possible values**: Any value between 10 and 32 000. The default value is 100. |
| Web max sessions | Longint | 71 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description**: Limits the number of simultaneous sessions. When you reach the limit set, the oldest session is closed (and **On Web Close Process database method** is called) if the Web server needs to create a new one.<br>**Possible values**: Longint. The number of simultaneous sessions cannot exceed the total number of Web processes (Web Max Concurrent Processes option, 100 by default)<br>**Default value**: 100 (pass 0 to restore the default value) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Web maximum requests size | Longint | 27 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Maximum size (in bytes) of incoming HTTP requests (POST) that the Web server is authorized to process. By default, the value is 2 000 000, i.e. a little less than 2 MB. Passing the maximum value (2 147 483 648) means that, in practice, no limit is set. This limit is used to avoid Web server saturation due to incoming requests that are too large. When a request reaches this limit, the 4D Web server refuses it.<br>**Possible values**: 500 000 to 2 147 483 648. |
| Web port ID | Longint | 15 | **Scope**: 4D in local mode and 4D Server.<br>**Kept between two sessions:** No<br>**Description:** Sets or gets the number of the TCP port used by the 4D Web server with 4D in local mode and 4D Server. By default, the value is 80. The TCP port number is set on the "Web/Configuration" page of the Database Settings dialog box. You can use one of the constants in the **TCP Port Numbers** theme for the *value* parameter. This selector is useful within the framework of 4D Web servers that are compiled and merged using 4D Desktop (no access to the Design environment).<br>**Possible values:** For more information about the TCP port number, refer to the **Web Server Settings** section.<br>**Default value:** 80 |
| Web session cookie domain | Longint | 81 | **Scope:** local Web server<br>**Kept between two sessions:** No, but remains valid even if the HTTP server is restarted.<br>**Description:** Sets or gets the value of the "domain" field of the session cookie. This selector (as well as selector 82) is useful for controlling the scope of the session cookies: If you set, for example, the value "/*.4d.fr" for this selector, the client will only send a cookie when the request is addressed to the domain ".4d.fr", which excludes servers hosting external static data.<br>**Possible values:** Text |
| Web session cookie name | Longint | 73 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description:** Sets the name of the cookie used for saving the session ID.<br>**Possible values:** Text<br>**Default value:** "4DSID" (pass an empty string to restore the default value) |
| Web session cookie path | Longint | 82 | **Scope:** local Web server<br>**Kept between two sessions:** No, but remains valid even if the HTTP server is restarted.<br>**Description:** Sets or gets the value of the "path" field of the session cookie. This selector (as well as selector 81) is useful for controlling the scope of the session cookies: If you set, for example, the value "/4DACTION" for this selector, the client will only send a cookie for dynamic requests beginning with 4DACTION, and not for pictures, static pages, etc.<br>**Possible values:** Text |
| Web session enable IP address validation | Longint | 83 | **Scope**: Local Web server<br>**Kept between two sessions**: No<br>**Description**: Enables or disables IP address validation for session cookies. For security reasons, by default the 4D Web server checks the IP address of each request containing a session cookie and rejects it if this address does not match the IP address used to create the cookie. In some specific applications, you may want to disable this validation and accept session cookies, even when their IP addresses do not match. For example when mobile devices switch between Wifi and 3G/4G networks, their IP address will change. In this case, you must pass 0 in this option to allow clients to be able to continue using their Web sessions even when the IP addresses change. Note that this setting lowers the security level of your application.<br>When it is modified, this setting is effective immediately (you do not need to restart the HTTP server).<br>**Possible values**: 0 (disabled) or 1 (enabled)<br>**Default value**: 1 (IP addresses are checked) |

When you use the Web debug log *selector*, you can pass one of the following constants in the *value* parameter:

| Constant | Type | Value | Comment |
|---|---|---|---|
| wdl disable | Longint | 0 | Web HTTP debug log is disabled |
| wdl enable with all body parts | Longint | 7 | Web HTTP debug log is enabled with body parts in response and request |
| wdl enable with request body | Longint | 5 | Web HTTP debug log is enabled with body part in request only |
| wdl enable with response body | Longint | 3 | Web HTTP debug log is enabled with body part in response only |
| wdl enable without body | Longint | 1 | Web HTTP debug log is enabled without body parts (body size is provided in this case) |

## Example

Enabling the HTTP debug log without body parts:

```
WEB SET OPTION(Web debug log;wdl enable without body)
```

A log entry looks like this:

```
# REQUEST
# SocketID: 1592
# PeerIP: 127.0.0.1
# PeerPort: 54912
# TimeStamp: 39089388
GET /4DWEBTEST HTTP/1.1
Connection: Close
Host: 127.0.0.1
User-Agent: 4D_HTTP_Client/0.0.0.0

# RESPONSE
# SocketID: 1592
# PeerIP: 127.0.0.1
# PeerPort: 54912
# TimeStamp: 39089389 (elapsed time: 1 ms)
HTTP/1.1 200 OK
Accept-Ranges: bytes
Connection: close
Content-Length: 3555
Content-Type: text/plain; charset=UTF-8
Date: Tue, 20 Jan 2015 10:51:29 GMT
Expires: Tue, 20 Jan 2015 10:51:29 GMT
Pragma: no-cache
Server: 4D/14.6.0

[Body Size: 3555]
```

# WEB SET ROOT FOLDER

WEB SET ROOT FOLDER ( rootFolder )

| Parameter | Type | | Description |
|---|---|---|---|
| rootFolder | String | ⇒ | Pathname of Web server root folder |

## Description

The **WEB SET ROOT FOLDER** command is used to modify the default root folder where 4D looks for the HTML files requested of the Web server.

This command does not take the default root folder that may have been set in the Database Settings into account. For more information about this folder, please refer to the **Connection Security** section.

The location of the root folder can be expression either in HTML syntax (URL type), or in system syntax (absolute path):

- HTML syntax: folder names are separated by a slash ("/"), regardless of the platform you use.
- System syntax: absolute pathname (“long name”) respecting the syntax of the current platform, for example:
  - (Mac OS) Disk:Applications:myserv:folder
  - (Windows) C:¥Applications¥myserv¥folder

**Notes:**

- The Web server will need to be restarted in order for the new root folder to be taken into account.
- You can find out the location of the current root folder at any time using the **Get 4D folder** command.

If you specify an invalid pathname, an OS File manager error is generated. You can intercept the error with an **ON ERR CALL** method. If you display an alert or a message from within the error method, it will appear on the browser side.

# WEB START SERVER

WEB START SERVER
Does not require any parameters

## Description

The **WEB START SERVER** command starts the Web server of the 4D application on which it has been executed (4D or 4D Server). The database is therefore published on your Intranet network or on the Internet.

If the Web Server is successfully started, **OK** is set to *1*, otherwise **OK** is set to *0* (zero). For example, if the TCP/IP network protocol is not properly configured, **OK** is set to *0*.

## System variables and sets

If the Web Server is successfully started, **OK** is set to 1; otherwise **OK** is set to 0.

# WEB STOP SERVER

WEB STOP SERVER
Does not require any parameters

## Description

The **WEB STOP SERVER** command stops the Web server of the 4D application on which it has been executed (4D ou 4D Server). If the Web server has been started, all Web connections are stopped, and all Web processes terminated.

If the Web server has not been started, the command does nothing.

# WEB Validate digest

WEB Validate digest ( userName ; password ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| userName | Text | ➡ | User name |
| password | Text | ➡ | User password |
| Function result | Boolean | ⮌ | True=Authentication OK, False=Authentication failed |

## Description

The **WEB Validate digest** command checks the validity of the identifying information (name and password) provided by a user connecting to the Web server. This command must be used in the **On Web Authentication Database Method** in the context of Web authentication in Digest mode (see the **Connection Security** section).

In the *userName* and *password* parameters, pass the identifying information of the user stored locally. The command uses this information to generate a value that it compares with the information sent by the Web browser.

If the values are the same, the command returns True. Otherwise, it returns False.

You can use this mechanism to manage and maintain your own secure access system to the Web server by programming. Note that Digest validation cannot be used jointly with 4D passwords.

**Note:** If the browser does not support Digest authentication, an error is returned (authentication error).

## Example

Example using **On Web Authentication Database Method** in Digest mode:

```
  ` On Web Authentication Database Method
C_TEXT($1;$2;$5;$6;$3;$4)
C_TEXT($user)
C_BOOLEAN($0)
$0:=False
$user:=$5
 `For security reasons, refuse names containing @
If(WithWildcard($user))
   $0:=False
 `The WithWildcard method is described in the "On Web Authentication Database Method" section
Else
   QUERY([WebUsers];[WebUsers]User=$user)
   If(OK=1)
      $0:=WEB Validate digest($user;[WebUsers]password)
   Else
      $0:=False `User does not exist
   End if
End if
```

# _o_SET CGI EXECUTABLE

_o_SET CGI EXECUTABLE ( url1 {; url2} )

| Parameter | Type | | Description |
|---|---|---|---|
| url1 | String | ➡ | Access URL |
| url2 | String | ➡ | Access URL |

## Command disabled

This command is disabled starting with 4D v13. If it is called, the error 33 ("Method  or function not implemented") is generated.

# _o_SET WEB DISPLAY LIMITS

_o_SET WEB DISPLAY LIMITS ( numberRecords {; numberPages {; picRef}} )

| Parameter | Type | | Description |
|---|---|---|---|
| numberRecords | Longint | ⇒ | *** Disabled *** |
| numberPages | Longint | ⇒ | *** Disabled *** |
| picRef | Longint | ⇒ | *** Disabled *** |

## Command disabled

This command is disabled starting with 4D v13. If it is called, the error 33 ("Method  or function not implemented") is generated.

# _o_SET WEB TIMEOUT

_o_SET WEB TIMEOUT ( timeout )

| Parameter | Type | | Description |
|---|---|---|---|
| timeout | Longint | ⇨ | *** Disabled *** |

## Command disabled

This command is disabled starting with 4D v13. If it is called, the error 33 ("Method  or function not implemented") is generated.

# _o_Web Context

_o_Web Context -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | ⮌ | Always returns False |

## Command disabled

This command is disabled starting with 4D v13. If it is called, the error 33 ("Method  or function not implemented") is generated.

# Web Services (Client)

- Web Services (Client) Commands
- WEB SERVICE AUTHENTICATE
- WEB SERVICE CALL
- WEB SERVICE Get info
- WEB SERVICE GET RESULT
- WEB SERVICE SET OPTION
- WEB SERVICE SET PARAMETER

# Web Services (Client) Commands

Starting with version 2003, 4D supports“Web Services”, meaning that the program enables you to publish (server part) and/or use (client part) Web Services directly from your database. A Web Service is a set of functions published on a network. These functions can be called and used by any application compatible with Web Services and connected to a network. Web Services can carry out all types of tasks, such as supervising the routing of packages at a transporter’s, e-commerce, monitoring market values, etc.

For more information about the concept and operation of Web Services, refer to the Design Reference manual.

Subscription to Web Services with 4D is easy to carry out using the Web Services Wizard. In most cases, this Wizard will be sufficient for you to be able to use Web Services. However, if you want to customize certain mechanisms, you must use the client SOAP commands of 4D.

This section describes the commands used for subscription to external Web Services (**client part**). For a description of the commands used for the publication of Web Services (server part), refer to the **Web Services (Server) Commands** theme.

**Note**: By convention, the terms “SOAP” and “Web Service” have been used to differentiate between command (and constant) names on the server and client side, respectively. These two concepts refer to the same technology.

# WEB SERVICE AUTHENTICATE

WEB SERVICE AUTHENTICATE ( name ; password {; authMethod} {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| name | String | ⇒ | User name |
| password | String | ⇒ | User password |
| authMethod | Longint | ⇒ | Authentication method 0 or omitted = not specified, 1 = BASIC, 2 = DIGEST |
| * | Operator | ⇒ | If passed: authentication by proxy |

## Description

The **WEB SERVICE AUTHENTICATE** command enables the use of Web Services requiring authentication of the client application (simple authentication). The BASIC and DIGEST methods are supported, as well as the presence of a proxy.

**Note:** For more information about the BASIC and DIGEST authentication methods, refer to the **Connection Security** section.

In the *name* and *password* parameters, pass the required identification information (user name and password). This information will be encoded and added to the HTTP request sent to the Web Service using the **WEB SERVICE CALL** command. It is thus necessary to call the **WEB SERVICE AUTHENTICATE** command before calling the **WEB SERVICE CALL** command.

The optional *authMethod* parameter indicate the authentication method to be used for the next call to the **WEB SERVICE CALL** command. You can pass one of the following values:

- 2 = use the DIGEST authentication method
- 1 = use the BASIC authentication method
- 0 (or parameter omitted) = use the appropriate method. In this case, 4D sends an additional request in order to negotiate the authentication method.

If you pass the * parameter, you indicate that the authentication information is to be sent to an HTTP proxy. This configuration must be implemented when there is a proxy that requires authentication between the Web Service client and the Web Service itself. If the Web Service is itself authenticated, a double authentication is required (see the example).

By default, the authentication information is reset to zero after each request. Therefore, you must use the **WEB SERVICE AUTHENTICATE** command before each **WEB SERVICE CALL** command. It is nevertheless possible to keep this information temporarily using an option of the **WEB SERVICE SET OPTION** command. In this case, it is not necessary to execute the **WEB SERVICE AUTHENTICATE** command before each **WEB SERVICE CALL** command.

If authentication fails, the SOAP server returns an error that you can identify using the **WEB SERVICE Get info** command.

## Example

Authentication with a Web Service located behind a proxy:

```
 // Authentication to Web Service in DIGEST mode
WEB SERVICE AUTHENTICATE("SoapUser";"123";2)
 // Authentication to proxy in default mode
WEB SERVICE AUTHENTICATE("ProxyUser";"456";*)
WEB SERVICE CALL(...)
```

# WEB SERVICE CALL

WEB SERVICE CALL ( accessURL ; soapAction ; methodName ; nameSpace {; complexType {; *}} )

| Parameter | Type | | Description |
|---|---|---|---|
| accessURL | String | ⇒ | Access URL to Web Service |
| soapAction | String | ⇒ | Contents of SOAPAction field |
| methodName | String | ⇒ | Name of the method |
| nameSpace | String | ⇒ | Namespace |
| complexType | Longint | ⇒ | Configuration of complex types (simple types if omitted) |
| * | Operator | ⇒ | Do not close connection |

## Description

The **WEB SERVICE CALL** command calls a Web Service by sending an HTTP request. This request contains the SOAP message created previously using the **WEB SERVICE SET PARAMETER** command.

Any subsequent call to the **WEB SERVICE SET PARAMETER** command will cause the creation of a new request. The execution of the **WEB SERVICE CALL** command also erases any result from a previously-called Web Service and replaces it with the new result(s).

In *accessURL*, pass the complete URL allowing access to the Web Service (do not confuse this URL with that of the WSDL file, which describes the Web Service).

- **Access in secure mode (SSL)**: If you want to use a Web Service in secure mode using SSL, pass https:// in front of the URL instead of http://. This configuration automatically enables connection in secure mode.
  Note that this command can use a server certificate (see the **HTTP SET CERTIFICATES FOLDER** command). If this certificate is not valid (expired or revoked), the OK system variable is set to 0 and error 901 "Invalid server certificate" is returned. You can intercept this error using an error-handling method installed by the **ON ERR CALL** command.

In *soapAction*, pass the contents of the SOAPAction field of the request. This field generally contains the value “ServiceName#MethodName”.

In *methodName*, pass the name of the remote method (belonging to the Web Service) that you want to execute.

In *namespace*, pass the XML namespace used for the SOAP request. For more information about XML namespaces, refer to the Design Mode manual of 4D.

The optional *complexType* parameter specifies the configuration of the Web Service parameters sent or received (defined using the **WEB SERVICE SET PARAMETER** and **WEB SERVICE GET RESULT** commands).

The value of the *complexType* parameter depends on the publication mode of the Web Service (DOC or RPC, see the Design Reference manual of 4D) and on its own parameters.
In *complexType*, you must pass one of the following constants, located in the theme **Web Services (Client)**:

| Constant | Type | Value |
|---|---|---|
| Web Service dynamic | Longint | 0 |
| Web Service manual | Longint | 3 |
| Web Service manual in | Longint | 1 |
| Web Service manual out | Longint | 2 |

Each constant corresponds to a Web Services “configuration”. A configuration represents the combination of a publication mode (RPC/DOC) and the types of parameters (input/output, simple or complex) implemented.

**Note:** Remember that the “input” or “output” characteristic of parameters is evaluated from the point of view of the proxy method/Web Service:

- an “input” parameter is a value passed to the proxy method and thus to the Web Service,
- an “output” parameter is returned by the Web Service and thus by the proxy method (generally via $0).

The following table shows all the possible configurations as well as the corresponding constants:

| | Input parameters | |
|---|---|---|
| **Output parameters** | **Simple** | **Complex** |
| **Simple** | Web Service dynamic (RPC mode) | Web Service manual in (RPC mode) |
| **Complex** | Web Service manual out (RPC mode) | Web Service manual (RPC or DOC mode) |

The five configurations described below can therefore be implemented. In all cases, 4D will handle the formatting of the SOAP request to be sent to the Web Service as well as its envelope. It is up to you to format the contents of this request according to the configuration used.

**Note:** Despite the fact that they are complex XML types, data arrays are handled by 4D as simple types.

### RPC mode, simple input and output

This configuration is the easiest to use. In this case, the *complexType* contains the Web Service dynamic constant or is omitted.

The parameters sent and responses received can be handled directly, without prior processing.

Refer to the example of the command **WEB SERVICE GET RESULT**.

### RPC mode, complex input and simple output

In this case, the *complexType* parameter contains the Web Service manual in constant. With this configuration, you must "manually" pass each XML source element in the form of a BLOB to the Web Service, using the **WEB SERVICE SET PARAMETER** command.

It is up to you to format the initial BLOB as a valid XML element. As its first element, this BLOB must contain the first apparent "child" element of the <Body> element of the final request.

**Example**

```
C_BLOB($1)
C_BOOLEAN($0)

WEB SERVICE SET PARAMETER("MyXMLBlob";$1)
WEB SERVICE CALL("http://my.domain.com/my_service";"MySoapAction";"TheMethod";"http://my.namespace.com/";Web Service manual in)
WEB SERVICE GET RESULT($0;"MyOutputVar";*)
```

### RPC mode, simple input and complex output

In this case, the *complexType* parameter contains the Web Service manual out constant. Each output parameter will be returned by the Web Service in the form of an XML element stored in a BLOB. You retrieve this parameter using the **WEB SERVICE GET RESULT** command. You can then parse the contents of the BLOB received using the XML commands of 4D.

**Example**

```
C_BLOB($0)
C_BOOLEAN($1)

WEB SERVICE SET PARAMETER("MyInputVar";$1)
WEB SERVICE CALL("http://my.domain.com/my_service";"MySoapAction";"TheMethod";"http://my.namespace.com/";Web Service manual out)
WEB SERVICE GET RESULT($0;"MyXMLOutput";*)
```

### RPC mode, complex input and output

In this case, the *complexType* parameter contains the Web Service manual constant. Each input and output parameter must be stored in the form of XML elements in BLOBs, as described in the two previous configurations.

**Example**

```
C_BLOB($0)
C_BLOB($1)
```

```
WEB SERVICE SET PARAMETER("MyXMLInputBlob";$1)
WEB SERVICE CALL("http://my.domain.com/my_service";"MySoapAction";"TheMethod";"http://my.namespace.com/";Web Service manual)
WEB SERVICE GET RESULT($0;"MyXMLOutput";*)
```

### DOC mode

A proxy calling method for a DOC Web Service is similar to a proxy calling method for an RPC Web Service using complex type input and output parameters.

The only difference between these two configurations lies at the level of the XML content of BLOB parameters sent and received. From 4D's point of view, the building and sending of the SOAP request are identical.

#### Example

```
C_BLOB($0)
C_BLOB($1)

WEB SERVICE SET PARAMETER("MyXMLInput";$1)
WEB SERVICE CALL("http://my.domain.com/my_service";"MySoapAction";"TheMethod";"http://my.namespace.com/";Web Service manual)
WEB SERVICE GET RESULT($0;"MyXMLOutput";*)
```

**Note:** In the case of DOC Web Services, the value of the strings ("MyXMLInput" and "MyXMLOutput" above) passed as parameters is of no importance; it is even possible to pass empty strings "". In fact, these values are not used in the SOAP request containing the XML document. It is, nevertheless, mandatory to pass these parameters.

To use a Web Service published in DOC mode (or in RPC mode with complex types), it is advisable to proceed as follows:

- Generate the proxy method using the Client Web Services Wizard.
  The proxy method will be called in the following manner: *$XMLresultBlob:=$DOCproxy_Method($XMLparamBlob)*
- Familiarize yourself with the contents of SOAP requests to be sent to the Web Service using an on-line test (for instance, *http://soapclient.com/soaptest.html*). This type of tool is used to generate HTML test forms based on the WSDL of the Web Service.
- Copy the XML contents generated from the first child element of *<body>*.
- Write the method enabling you to place the real parameter values into the XML code; this code must then be placed in the *$XMLparamBlob* BLOB.
- To parse the response, you can also use an on-line test, or make use of the WSDL that specifies the returned elements.

The * parameter can be used to optimize calls. When it is passed, the command does not close the connection used by the process at the end of its execution. In this case, the next call to **WEB SERVICE CALL** will reuse this same connection if the * parameter is passed, and so on. To close the connection, simply execute the command without the * parameter. This mechanism can be used to noticeably accelerate the processing of successive calls of several different Web Services on the same server, in particular in a WAN configuration (via the Internet, for example). Note that this mechanism depends on the "keep-alive" setting of the Web server. This setting generally defines a maximum number of requests via the same connection, and can even deny requests. If the **WEB SERVICE CALL** requests following each other in the same connection reach this maximum number, or if keep-alive connections are not allowed, 4D will create a new connection for each request.

## System variables and sets

If the request has been correctly routed and the Web Service has accepted it, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is returned.

# WEB SERVICE Get info

WEB SERVICE Get info ( infoType ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| infoType | Longint | ⇒ | Information to be retrieved |
| Function result | String | ⇐ | Information about the last SOAP error |

## Description

The **WEB SERVICE Get info** command returns information about any error generated during the execution of the last SOAP request sent to a remote Web Service. This command should generally be called within an error-handling method installed by the **ON ERR CALL** command.

The *infoType* parameter allows you to indicate the type of information that you want to obtain. You must pass one of the constants listed below, located in the **Web Services (Client)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Web Service detailed message | Longint | 1 | Detailed message describing the error. The type of message differs according to the main error type.<br>- If the main error = 9910 (Soap fault): the cause of the SOAP fault is returned (e.g.: “the remote method does not exist”).<br>- If the main error = 9911 (Parser fault): the location of the error in the XML document is returned.<br>- If the main error = 9912 (HTTP fault):<br>- if the HTTP error is located in the interval [300-400] (problems linked to the location of the requested document), the new location of the requested URL is returned.<br>- for any other HTTP error code, the <body> is returned.<br>- If the main error = 9913 (Network fault): the cause of the network fault is returned (e.g.: “ServerAddress: DNS lookup failure”)<br>- If the main error = 9914 (Internal fault): the cause of the internal fault is returned |
| Web Service error code | Longint | 0 | Main error code (defined by 4D). This code is also returned in the Error system variable.<br>List of codes that may be returned:<br>9910: Soap fault (see also Web Service Fault Actor)<br>9911: Parser fault<br>9912: HTTP fault (see also Web Service HTTP Error code)<br>9913: Network fault<br>9914: Internal fault. |
| Web Service fault actor | Longint | 3 | Cause of the error (returned by the SOAP protocol — to be used in the case of main error 9910).<br>- Version Mismatch<br>- Must Understand (the server was unable to interpret a parameter defined as mandatory)<br>- Sender Fault<br>- Receiver Fault<br>- Encoding Unknown |
| Web Service HTTP error code | Longint | 2 | HTTP error code (to be used in case of main error 9912). |

An empty string is returned when no information is available, more particularly when the last SOAP request did not generate any errors.

# WEB SERVICE GET RESULT

WEB SERVICE GET RESULT ( returnValue {; returnName {; *}} )

| Parameter | Type | | Description |
|---|---|---|---|
| returnValue | Variable | ⇐ | Value returned by the Web Service |
| returnName | String | ⇒ | Name of the parameter to be retrieved |
| * | | ⇒ | Free up memory |

## Description

The **WEB SERVICE GET RESULT** command retrieves a value sent back by the Web Service as a result of the processing performed.

**Note:** This command must be used only after the **WEB SERVICE CALL** command.

The *returnValue* parameter receives the value sent back by the Web Service. Pass a 4D variable in this parameter. This variable is generally $0, corresponding to the value returned by the proxy method. It is, however, possible to use intermediary variables (you must use process variables only).

**Note:** Each 4D variable or array used must be previously declared using the commands of the "**Compiler** and **Arrays**" themes.

The optional *returnName* parameter specifies the name of the parameter to be retrieved. However, since most Web Services only return a single value, this parameter is generally not necessary.

The optional * parameter signals the program to free up the memory devoted to the processing of the request. You must pass this parameter after retrieving the last value sent by the Web Service.

## Example

Imagine that a Web Service returns the current time in any city in the world. The parameters received by the Web Service are the name of the city and the country code. In return, the Web Service sends the corresponding time. The proxy calling method could be in the following form:

```
C_TEXT($1)
C_TEXT($2)
C_TIME($0)

WEB SERVICE SET PARAMETER("city";$1)
WEB SERVICE SET PARAMETER("country_code";$2)

WEB SERVICE CALL("http://www.citiesoftheworld.com/WS";"WSTime#City_time";"City_time";
"http://www.citiesoftheworld.com/namespace/default")

If(OK=1)
   WEB SERVICE GET RESULT($0;"return";*)
End if
```

# WEB SERVICE SET OPTION

WEB SERVICE SET OPTION ( option ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| option | Longint | ⇨ | Code of the option to set |
| value | Longint, Text | ⇨ | Value of the option |

## Preliminary note

This command is designed for advanced Web Services users. Its use is optional.

## Description

The **WEB SERVICE SET OPTION** command sets different options to be used during the next SOAP request triggered using the **WEB SERVICE CALL** command.
You can call this command as many times as there are options to be set.

In the *option* parameter, pass the number of the option to set and in the *value* parameter, pass the new value of the option. For these parameters, you can use one of the following predefined constants of the **Web Services (Client)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Web Service display auth dialog | Longint | 4 | *value* = 0 (do not display dialog box) or 1 (display dialog box)<br>This option manages the display of the authentication dialog box during execution of the **WEB SERVICE CALL** command. By default, this command never displays the dialog box; normally, you have to use the **WEB SERVICE AUTHENTICATE** command to do so. However, if you want the authentication dialog box to appear in order for the user to enter their identifiers, you will need to use this option: pass 1 in *value* to display the dialog box and 0 otherwise. The dialog box only appears if the Web service requires authentication |
| Web Service HTTP compression | Longint | 6 | *value* = Web Service Compression<br>This option is used to enable an internal compression mechanism for SOAP requests in order to accelerate inter-4D application exchanges. When you execute the statement **WEB SERVICE SET OPTION**(Web Service HTTP Compression; Web Service Compression) on the 4D client of the Web Service, the data of the next SOAP request sent by the client will be compressed using a standard HTTP mechanism ("gzip" or "deflate", depending on the request contents) before being sent to the 4D SOAP server. The server will decompress and parse the request, then will reply automatically using the same mechanism. Only the request that follows the call to the **WEB SERVICE SET OPTION** command is affected. You must therefore call this command each time you want to use compression. By default, 4D does not compress Web Service HTTP requests.<br>**Note:** This mechanism cannot be used for requests sent to a 4D SOAP server whose version is earlier than 11.3. So that you can further optimize this functioning, additional options configure the threshold and compression rate of the requests. These options can be accessed via the **SET DATABASE PARAMETER** command |
| Web Service HTTP timeout | Longint | 1 | *value* = timeout of the client portion expressed in seconds.<br>The timeout of the client portion is the wait period of the Web Service client in case the server does not respond. After this period, the client closes the session and the request is lost.<br>This timeout is 180 seconds by default. It can be modified for specific reasons (network status, Web Service specifics, etc.). |
| Web Service reset auth settings | Longint | 5 | *value* = 0 (do not erase information) or 1 (erase information)<br>This option lets you indicate to 4D whether to memorize the authentication information of the user (user name, password, method, etc.), in order to reuse it subsequently. By default, this information is erased after each execution of the **WEB SERVICE CALL** command. Pass 0 in value to store the information and 1 to erase it. Note that when you pass 0, the information is kept during the session but is not stored. |
| Web Service SOAP header | Longint | 2 | *value* = XML root element reference to insert as a header in the SOAP request.<br>This option allows you to insert a header in a SOAP request generated using the **WEB SERVICE CALL** command. SOAP requests do not contain a specific header by default. However, certain Web Services require a header, for example when managing identification parameters |
| Web Service SOAP version | Longint | 3 | *value* = Web Service SOAP 1 1 or Web Service SOAP 1 2<br>This option lets you specify the SOAP protocol version used in the request. Pass the Web Service SOAP 1 1 constant in value to indicate version 1.1 and Web Service SOAP 1 2 to indicate version 1.2. |

The order in which the options are called is not important. If the same *option* is set several times, only the value of the last call is taken into account.

## Example 1

Inserting a customized header in the SOAP request:

```
  ` Creating an XML reference
C_TEXT(vRootRef;vElemRef)
vRootRef:=DOM Create XML Ref("RootElement")
vxPath:="/RootElement/Elem1/Elem2/Elem3"
vElemRef:=DOM Create XML element(vRootRef;vxPath)
  `Modifying SOAP header with reference
WEB SERVICE SET OPTION(Web Service SOAP header;vElemRef)
```

## Example 2

Using version 1.2 of the SOAP protocol:

```
WEB SERVICE SET OPTION(Web Service SOAP version;Web Service SOAP_1_2)
```

# WEB SERVICE SET PARAMETER

WEB SERVICE SET PARAMETER ( name ; value {; soapType} )

| Parameter | Type | | Description |
|---|---|---|---|
| name | String | ➡ | Name of parameter to include in SOAP request |
| value | Variable | ➡ | 4D variable containing the value of the parameter |
| soapType | String | ➡ | SOAP type of the parameter |

## Description

The **WEB SERVICE SET PARAMETER** command enables the definition of a parameter used for a client SOAP request. Call this command for each parameter in the request (the number of times the command is called depends on the number of parameters).

In *name*, pass the name of the parameter as it must appear in the SOAP request.

In *value*, pass the 4D variable containing the value of the parameter. In the case of proxy methods, this variable is generally *$1*, *$2*, *$3*, etc., corresponding to a 4D parameter passed to the proxy method when it was called. It is, however, possible to use intermediary variables.

**Note:** Each 4D variable or array used must first be declared using the commands of the theme **Compiler** or **Arrays**.

By default, 4D automatically determines the most appropriate SOAP type for the *name* parameter according to the content of *value*. The indication of the type is included in the request.

However, you may want to “force” the definition of the SOAP type of a parameter. In this case, you can pass the optional *soapType* parameter using one of the following character strings (primary data types):

| soapType | Description |
|---|---|
| string | String |
| int | Longint |
| boolean | Boolean |
| float | 32-bit real |
| decimal | Real with decimal |
| double | 64-bit real |
| duration | Duration in years, months, days, hours, minutes, seconds, for example P1Y2M3DT10H30M |
| datetime | Date and time in ISO8601 format, for example 2003-05-31T13:20:00 |
| time | Time, for example 13:20:00 |
| date | Date, for example 2003-05-31 |
| gyearmonth | Year and month (Gregorian calender), for example 2003-05 |
| gyear | Year (Gregorian calender), for example 2003 |
| gmonthday | Month and day (Gregorian calender), for example --05-31 |
| gday | Day (Gregorian calender), for example ---31 |
| gmonth | Month (Gregorian calender), for example --10-- |
| hexbinary | Value expressed in hexadecimal |
| base64binary | BLOB |
| anyuri | Uniform Resource Identifier (URI), for example http://www.company.com/page |
| qname | Qualified XML name (namespace and local part) |
| notation | Notation attribute |

**Note:** For more information about XML data types, refer to the URL *http://www.w3.org/TR/xmlschema-2/*

## Example

This example defines two parameters:

```
C_TEXT($1)
C_TEXT($2)
WEB SERVICE SET PARAMETER("city";$1)
WEB SERVICE SET PARAMETER("country";$2)
```

# Web Services (Server)

- Web Services (Server) Commands
- SOAP DECLARATION
- SOAP Get info
- SOAP Request
- SOAP SEND FAULT

## Web Services (Server) Commands

---

Publication of Web Services with 4D is carried out easily using options in the method properties. In most cases, this operation will be sufficient to enable you to publish Web Services. However, if you want to customize certain mechanisms, use data arrays, etc., you must use the server SOAP commands of 4D.

This section describes the commands used for the publication of Web Services in 4D (**server part**). For more general information about Web Services or for a description of the commands used for subscription to Web Services (client part), refer to the **Web Services (Client) Commands** section.

**Note**: By convention, the terms “SOAP” and “Web Service” have been used to differentiate between command (and constant) names on the server and client side, respectively. These two concepts refer to the same technology.

# SOAP DECLARATION

SOAP DECLARATION ( variable ; type ; input_output {; alias} )

| Parameter | Type | | Description |
|---|---|---|---|
| variable | Variable | ⇒ | Variable referring to an incoming or outgoing SOAP argument |
| type | Longint | ⇒ | 4D type to which the argument points |
| input_output | Longint | ⇒ | 1 = SOAP Input, 2 = SOAP Output |
| alias | String | ⇒ | Name published for this argument during SOAP exchanges |

## Description

The **SOAP DECLARATION** command explicitly declares the type of parameters used in a 4D method published as a Web Service.

When a method is published as a Web Service, the standard parameters *$0*, *$1*... *$n* describe the parameters of the Web Service to the outside world (more particularly in the WSDL file). The SOAP protocol requires that parameters be explicitly named; 4D uses the names “FourD_arg0, FourD_arg1 ... FourD_argn” by default.

This default operation can nevertheless prove to be problematic for the following reasons:

- It is not possible to declare *$0* or *$1*, *$2*, etc. as an array. Therefore, it is necessary to use pointers; however, in this case, the type of values must be known for the generation of the WSDL file.
- It can be useful or necessary to customize the parameter names (incoming and outgoing).
- You may want to use parameters such as XML structures and DOM references.
- It may also be necessary to retur values with a size greater than 32 KB (limit for Text arguments in databases that are in non-Unicode mode).
- Finally, this operation makes it impossible to return more than one value per RPC call (in *$0*).

The **SOAP DECLARATION** command lets you be free from these limits. You can execute this command for each incoming and outgoing parameter to assign it a name and a type.

**Note:** Even when the this command is used, it is still necessary to declare 4D variables and arrays in the Compiler_Web method using commands of the “**Compiler**” theme.

In *variable*, pass the 4D variable to be referred to when calling the Web Service.

**Warning:** You can only refer to process variables or 4D method arguments (*$0* to *$n*). Local and interprocess variables cannot be used.

By default, because only Text type arguments can be used, the SOAP server responses are limited to 32 KB in databases that are in non-Unicode mode. However, it is possible to return SOAP arguments with a size greater than 32 KB, using BLOBs. To exceed this limit, you simply need to declare the arguments as BLOBs before calling the **SOAP DECLARATION** command (see example 4).

**Note:** On the client side, if you subscribe to this type of Web Service with 4D, the Web Services Wizard will of course generate a Text type variable. To be able to use it, you just need to re-type this return variable as a BLOB in the proxy method.

In *type*, pass the corresponding 4D type. Most types of 4D variables and arrays can be used. You can use the following predefined constants, located in the “**Field and Variable Types**” theme, as well as, for XML types, two constants of the **Web Services (Server)** theme:

| Constant | Type | Value |
|---|---|---|
| Boolean array | Longint | 22 |
| Date array | Longint | 17 |
| Integer array | Longint | 15 |
| Is BLOB | Longint | 30 |
| Is Boolean | Longint | 6 |
| Is date | Longint | 4 |
| Is integer | Longint | 8 |
| Is longint | Longint | 9 |
| Is real | Longint | 1 |
| Is string var | Longint | 24 |
| Is text | Longint | 2 |
| Is time | Longint | 11 |
| LongInt array | Longint | 16 |
| Real array | Longint | 14 |
| String array | Longint | 21 |
| Text array | Longint | 18 |

| Constant | Type | Value |
|---|---|---|
| Is DOM reference | Longint | 37 |
| Is XML | Longint | 36 |

In *input_output*, pass a value indicating whether the processed parameter is “incoming” (i.e. corresponding to a value received by the method) or “outgoing” (i.e. corresponding to a value returned by the method). You can use the following predefined constants, located in the **Web Services (Server)** theme:

| Constant | Type | Value |
|---|---|---|
| SOAP input | Longint | 1 |
| SOAP output | Longint | 2 |

### Use of XML types

You can declare variables of the "XML structure" and "DOM reference" type, both incoming and outgoing, via the Is XML and Is DOM reference constants. When parameters of this type are specified, no processing or encoding is applied to them and the data are transmitted "as is" (see example 5).

- Outgoing parameters:
  - Is XML indicates that the parameter contains an XML structure,
  - Is DOM reference indicates that the parameter contains the DOM reference of an XML structure. In this case, inserting the XML structure into the SOAP message is equivalent to executing the **DOM EXPORT TO VAR** command.

**Note:** In the case of DOM references used as outgoing parameters, it is recommended to use global references, created, for example, on startup and closed when the application is closed. In fact, a DOM reference created within the Web Service itself cannot be closed with **DOM CLOSE XML**, otherwise the Web Service no longer returns anything. Multiple calls to the Web Service therefore involve creating multiple unclosed DOM references, which can lead to memory saturation

- Incoming parameters:
  - Is XML indicates that the parameter must receive an XML argument sent by the SOAP client.
  - Is DOM reference indicates that the parameter must receive the DOM reference of an XML structure corresponding to the XML argument sent by the SOAP client.

- Modification of the WSDL: These XML structures will be declared by 4D as the "anyType" type (undetermined) in the WSDL. If you want to type an XML structure precisely, you must save the WSDL file and manually add the desired data schema in the <types> section of the WSDL

### COMPILER_WEB method

Incoming SOAP arguments referred to using 4D variables (and not 4D method arguments) must first be declared in the COMPILER_WEB project method. In fact, the use of process variables in Web Services methods requires that they be declared before the method is called. The COMPILER_WEB project method is called, if it exists, for each SOAP request accepted. By default, the COMPILER_WEB method does not exist. You must specifically create it.

Note that the COMPILER_WEB method is also called by the 4D Web server when receiving “conventional” Web requests of the POST type (see **URLs and Form Actions** section).

In *alias*, pass the name of the argument as it must appear in the WSDL and in the SOAP exchanges.

**Warning:** This name must be unique in the RPC call (both input and output parameters taken together), otherwise, only the last declaration will be taken into account by 4D.

**Note:** The argument names must not begin with a number nor contain spaces. Moreover, to avoid any risks of incompatibility, it is recommended to not use extended characters (such as accented characters).
If the *alias* parameter is omitted, 4D will use, by default, the name of the variable or *FourD_argN* for the 4D method arguments (*$0* to *$n*).

**Note:** The **SOAP DECLARATION** command must be included in the method published as a Web Service. It is not possible to call it from another method.

## Example 1

This example specifies a parameter name:

```
 ` In the COMPILER_WEB method
C_LONGINT($1)

 ` In the Web Service method
 ` During generation of the WSDL file and SOAP calls, the word
 ` zipcode will be used instead of fourD_arg1
SOAP DECLARATION($1;Is longint;SOAP input;"zipcode")
```

## Example 2

This example retrieves an array of zip codes in the form of longints:

```
 `In the COMPILER_WEB method
ARRAY LONGINT(codes;0)

 `In the Web service method
SOAP DECLARATION(codes;LongInt array;SOAP input;"in_codes")
```

## Example 3

This example refers to two return values without specifying an argument name:

```
SOAP DECLARATION(ret1;Is longint;SOAP output)
SOAP DECLARATION(ret2;Is longint;SOAP output)
```

## Example 4

This example allows the 4D SOAP server to return an argument with a size greater than 32 KB in databases in non-Unicode mode:

```
C_BLOB($0)
SOAP DECLARATION($0;Is text;SOAP output)
```

Note the type Is text (and not Is BLOB). This allows the argument to be correctly processed.

## Example 5

This example illustrates the results of different types of declarations:

```
ALL RECORDS([Contact])

 `Build an XML structure from the Contacts selection and store the XML in a BLOB
C_BLOB(ws_vx_xmlBlob)
```

```
getContactsXML(->ws_vx_xmlBlob)
 `Retrieve the XML structure in a text variable
C_TEXT(ws_vt_xml)
ws_vt_xml:=BLOB to text(ws_vx_xmlBlob;UTF8 text without length)
 `Retrieve a DOM reference to the XML structure
C_TEXT(ws_vt_xmlRef)
ws_vt_xmlRef:=DOM Parse XML variable(ws_vt_xml)

 `Test the various declarations
SOAP DECLARATION(ws_vx_xmlBlob;Is BLOB;SOAP output;"contactListsX")
 `The XML is converted into Base64 by 4D

SOAP DECLARATION(ws_vt_xml;Is text;SOAP output;"contactListsText")
 `The XML is converted into text by 4D (< > become entities)

SOAP DECLARATION(ws_vt_xml;Is XML;SOAP output;"xmlContacts")
 `The XML is passed as XML text

SOAP DECLARATION(ws_vx_xmlBlob;Is XML;SOAP output;"blobContacts")
 `The XML is passed as an XML BLOB

SOAP DECLARATION(ws_vt_xmlRef;Is DOM reference;SOAP output;"contactByRef")
 `The XML is passed as a reference
```

# SOAP Get info

SOAP Get info ( infoNum ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| infoNum | Longint | ⇒ | Number of type of SOAP info to get |
| Function result | String | ⇐ | SOAP Information |

## Description

The **SOAP Get info** command retrieves, in the form of a character string, the different types of information concerning a SOAP request.

When you process a SOAP request, it can be useful to obtain additional information — other than the RPC parameter values — about the request. For instance, for security reasons, you can use this command in the **On Web Authentication Database Method** to find out the name of the requested Web Service method.

Pass the number of the type of SOAP information you want to get in the *infoNum* parameter. You can use the following predefined constants, located in the **Web Services (Server)** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| SOAP method name | Longint | 1 | Name of the Web Service method about to be executed |
| SOAP service name | Longint | 2 | Name of the Web Service to which the method belongs |

**Note:** Also for security reasons, it is possible to set the maximum size for Web Services requests sent to 4D. This configuration is carried out using the **SET DATABASE PARAMETER** command.

# SOAP Request

SOAP Request -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Boolean | | True if the request is SOAP; otherwise, False |

## Description

The **SOAP Request** command returns **True** if the code being executed is part of a SOAP request.

This command can be used for security reasons in the **On Web Authentication Database Method** in order to determine the nature of the received requests.

# SOAP SEND FAULT

SOAP SEND FAULT ( faultType ; description )

| Parameter | Type | | Description |
|---|---|---|---|
| faultType | Longint | ⇒ | 1 = Client fault, 2 = Server fault |
| description | String | ⇒ | Description of error to be sent to SOAP client |

## Description

The **SOAP SEND FAULT** command returns an error to a SOAP client indicating the origin of the fault: client or server. Using this command lets you indicate an error to a client without having to return a result.

For instance, a fault on the client side may be detected when you publish a “Square_root” Web Service and a client sends a request with a negative number; you can use this command to indicate to the client that a positive value is required.

A possible fault on the server side may be, for instance, a lack of memory occurring during method execution.

Pass the origin of the error in *faultType*. You can use the following predefined constants, located in the **Web Services (Server)** theme:

| Constant | Type | Value |
|---|---|---|
| SOAP client fault | Longint | 1 |
| SOAP server fault | Longint | 2 |

Pass a description of the error in *description*. If the client implementation is in conformity, the error can be processed.

## Example

To go back to the example of the “Square_root” Web Service provided in the command description, the following command can be used to process requests with negative numbers:

```
SEND SOAP FAULT(SOAP client fault;"Positive values required")
```

# Windows

- Managing Windows
- Window Types
- CLOSE WINDOW
- CONVERT COORDINATES
- Current form window
- DRAG WINDOW
- ERASE WINDOW
- Find window
- Frontmost window
- GET WINDOW RECT
- Get window title
- HIDE TOOL BAR
- HIDE WINDOW
- MAXIMIZE WINDOW
- MINIMIZE WINDOW
- Next window
- Open form window
- Open window
- REDRAW WINDOW
- RESIZE FORM WINDOW
- SET WINDOW RECT
- SET WINDOW TITLE
- SHOW TOOL BAR
- SHOW WINDOW
- Tool bar height
- Window kind
- WINDOW LIST
- Window process
- *_o_Open external window*

# Managing Windows

Windows are used to display information to the user. They have three main uses: to enter data, to display data, and to inform the user in messages and dialogs.

There is always at least one window open. Scroll bars are added, when needed, to let the user scroll in a form that is larger than the window. In the Design environment, this window displays either the record list (output form) or the data entry screen (input form). In the Application environment, this window displays a splash screen (a custom graphic).

When you execute a menu command within the Application process, the splash screen can be replaced with data by commands that display forms. When the commands finish executing, the splash screen is displayed again by default.

## WinRef

You can open various types of custom windows with the **Open window** or **Open form window** commands (see the **Window Types** section). All windows opened by these commands are referenced through a **WinRef** expression. A *WinRef* is the unique ID of each open window. It is a Longint expression. All commands working with custom windows expect a *WinRef* parameter.

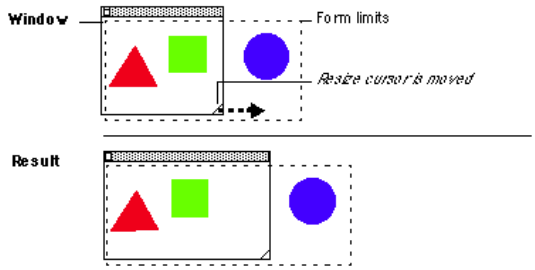
When you no longer need a custom window, you should close it using the **CLOSE WINDOW** command or by clicking the Control-menu box (Windows) or Close Box (Macintosh), if it exists.

Some commands open their own windows. Commands such as **QR REPORT** and **PRINT LABEL** open a window that becomes the frontmost window.

If you start a new process and do not open a window at the beginning of the process method, 4D will automatically open a default one as soon as a form is to be displayed.

## Side pushers

The right side and bottom of windows have become “pusher” splitters by default. This means that objects found to the right or below the limits of a window on screen are automatically pushed to the right or towards the bottom if the window is enlarged:




This mechanism allows you to manage retractable windows like the Explorer window (see the example of the **FORM SET SIZE** command).

**Note:** This does not work with windows that have scrollbars.

## Window coordinates and "right-to-left" mode

In window management commands, the window coordinates are determined with respect to a point of origin generally situated at the top left of the window/screen.
However, when the "right-to-left" mode is activated for the application, the coordinates are reversed and the point of origin switches to the top right of the window/screen. Consequently, in this mode the horizontal coordinates used by the following commands must also be reversed:

**Open window**
**Open form window**
**_o_Open external window**
**GET WINDOW RECT**

**SET WINDOW RECT**
**Find window**

**Note:** For more information about "right-to-left" mode, please refer to the Design Reference manual and to the description of the **SET DATABASE PARAMETER** command.

# Window Types

You can use one of the following predefined constants to specify the type of window that you open with **Open window**:

| Constant | Type | Value | Comment |
|---|---|---|---|
| Plain no zoom box window | Longint | 0 | |
| Modal dialog box | Longint | 1 | |
| Plain dialog box | Longint | 2 | Can be a floating window |
| Alternate dialog box | Longint | 3 | Can be a floating window |
| Plain fixed size window | Longint | 4 | |
| Movable dialog box | Longint | 5 | Can be a floating window |
| Plain window | Longint | 8 | |
| Round corner window | Longint | 16 | |
| Pop up window | Longint | 32 | |
| Sheet window | Longint | 33 | |
| Resizable sheet window | Longint | 34 | |
| Palette window | Longint | 1984 | Can be a floating window |

### Floating Windows

If you pass one of these constants to **Open window**, you open a regular windows. To open a floating windows, pass a negative window type value to **Open window**.

The main characteristic of floating windows is that they remain in the foreground even if the user clicks on another window of the process. Floating windows are generally used to display permanent information or tool bars.

### Modal windows

A modal window places the user in a state (or “mode”) where they can only act within this window. As long as the modal window is displayed, the menu commands and other application windows are inaccessible. To close a modal window, the user must either validate it, cancel it, or choose one of the options it offers. Warning dialog boxes are a typical example of modal windows.
In 4D, windows of the types 1 and 5 are modal windows.

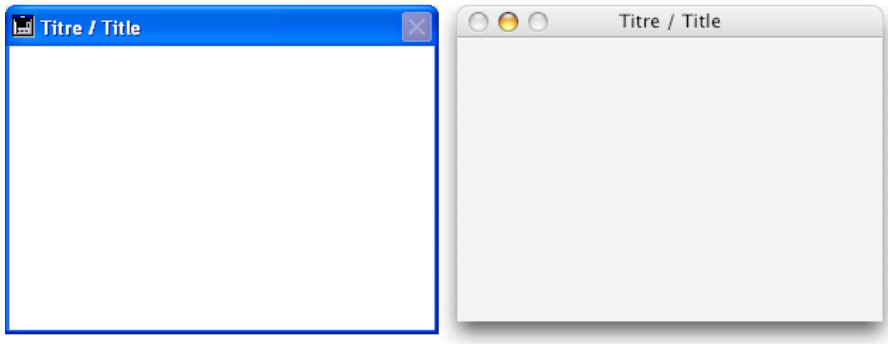
**Note:** A modal window always stays in the foreground. As a consequence, when a modal window calls a non-modal window, this latter window is displayed in the background, even though it was called subsequent to the modal window. You should thus avoid this type of operation.

On the other hand, when a modal window calls another modal window, this latter window will be displayed in the foreground.

## Description

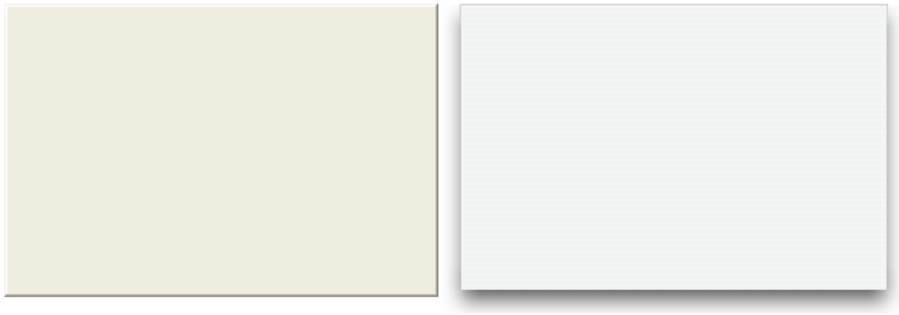
Here are the descriptions for each type of window under Windows (left) and Mac OS (right).

### Plain fixed size window (4)



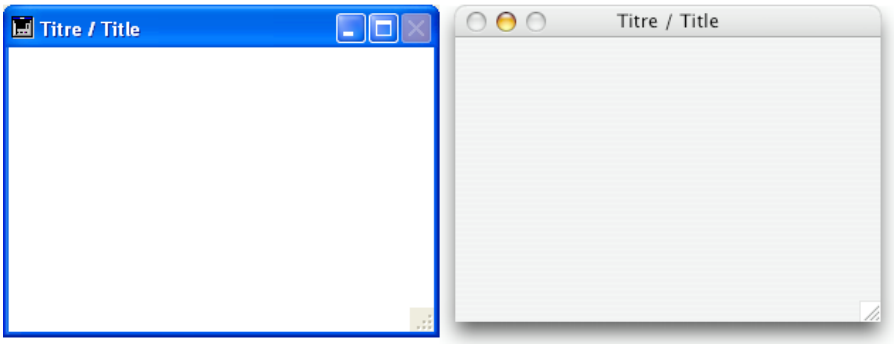
- Can have a title: Yes
- Can have a close box or equivalent: Yes
- Can be resized: No on Macintosh
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: Yes and No
- Usage: data entry with **ADD RECORD** or equivalent

### Modal dialog box (1)



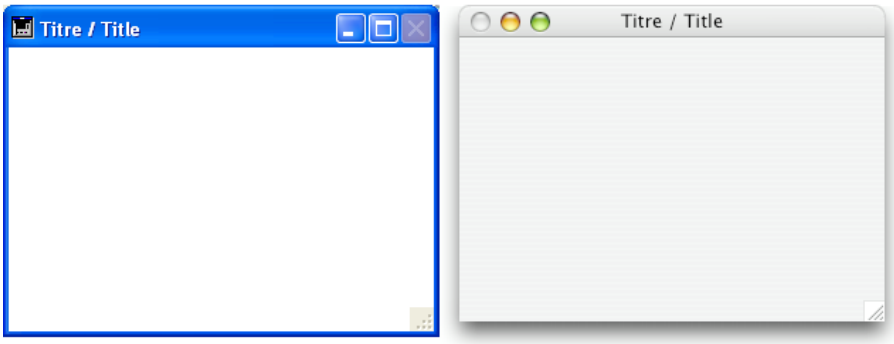
- Can have a title: No
- Can have a close box or equivalent: No
- Can be resized: No
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No
- Usage: **DIALOG**, **ADD RECORD** or equivalent
- Windows of this type are modal

### Plain no zoom box window (0)




- Can have a title: Yes
- Can have a close box or equivalent: Yes
- Can be resized: Yes
- Can be minimized/maximized or zoomed: No on Macintosh
- Suitable for scroll bars: Yes
- Usage: data entry with scrollbars, **DISPLAY SELECTION**, **MODIFY SELECTION**, etc.

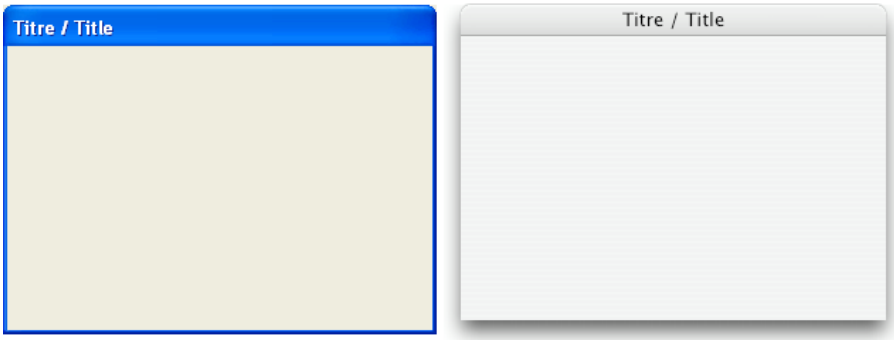
### Plain window (8)




- Can have a title: Yes
- Can have a close box or equivalent: Yes
- Can be resized: Yes
- Can be minimized/maximized or zoomed: Yes
- Suitable for scroll bars: Yes

- Usage: data entry with scrollbars, **DISPLAY SELECTION**, **MODIFY SELECTION**, etc.

### Movable dialog box (5)




- Can have a title: Yes
- Can have a close box or equivalent: No
- Can be resized: No
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No
- Usage: **DIALOG**, **ADD RECORD** or equivalent
- Windows of this type are modal, but can be moved and can be used as floating windows

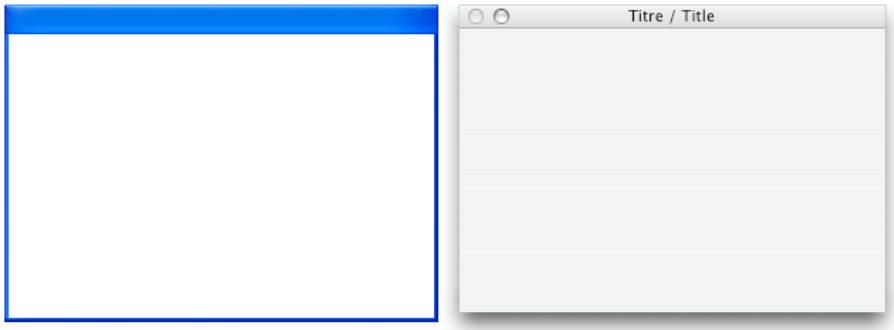
### Alternate dialog box (3)

- Can have a title: No
- Can have a close box or equivalent: No
- Can be resized: No
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No
- Usage: **DIALOG**, **ADD RECORD** or equivalent
- Windows of this type are modal, unless used as floating windows

### Plain dialog box (2)

- Can have a title: No
- Can have a close box or equivalent: No
- Can be resized: No
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No
- Usage: **DIALOG**, **ADD RECORD** or equivalent, splashscreens
- Windows of this type are modal, unless used as floating windows

### Palette window (1984)



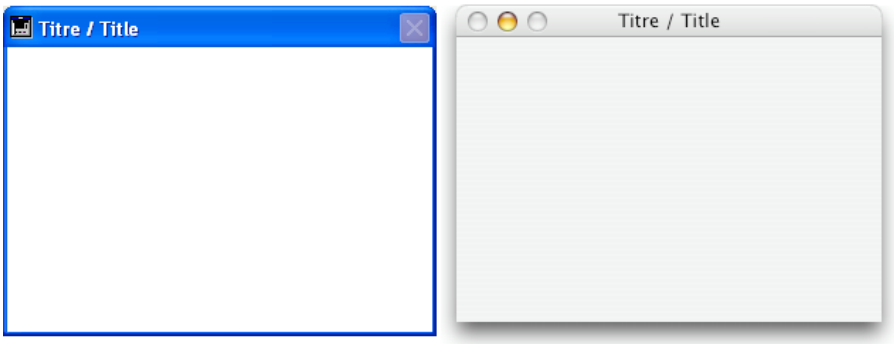

This type of window allows you to generate floating windows which can be defined as resizable or not. Only the following options are supported:

| Option | Value to pass under Windows | Value to pass under macOS |
|---|---|---|
| Not resizable | -(Palette window+2) | -Palette window |
| Resizable | -(Palette window+6) | -(Palette window+6) |

- Can have a title: Yes, if passed
- Can be resized: Yes, if the appropriate value is passed
- Usage: Floating windows with **DIALOG** or **DISPLAY SELECTION** (no data entry).

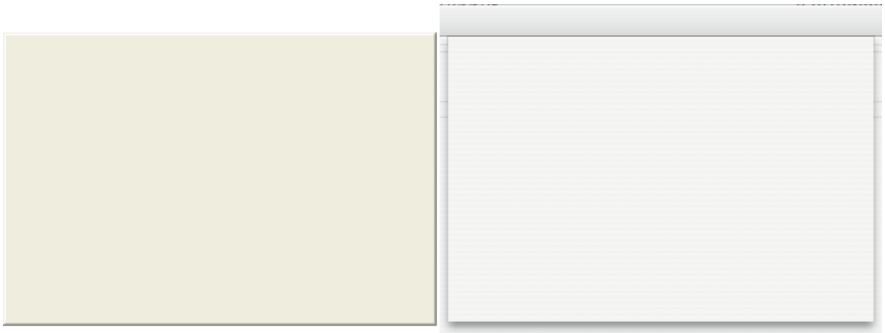
**Note:** With this type of window, the set of values (constant+option) must always be passed as a negative value. Make sure that you pass, for example, -(Palette window+6) and not (-Palette window+6).

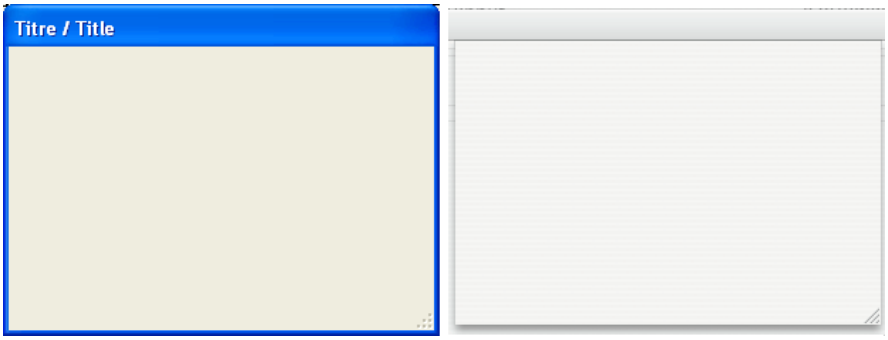
### Round corner window (16)




- Can have a title: Yes
- Can have a close box or equivalent: Yes
- Can be resized: No on Macintosh
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No on Macintosh
- Usage: Rare (obsolete)

### Sheet window (33) and Resizable sheet window (34)






Sheet windows are specific to Mac OS X. These windows “drop down” over the title bar of the main window using animation and are displayed above the main window. They are automatically centered in the main window. Their properties are comparable to those of the modal dialog boxes. They are generally used to perform an action directly relating to the action occurring in the primary window.

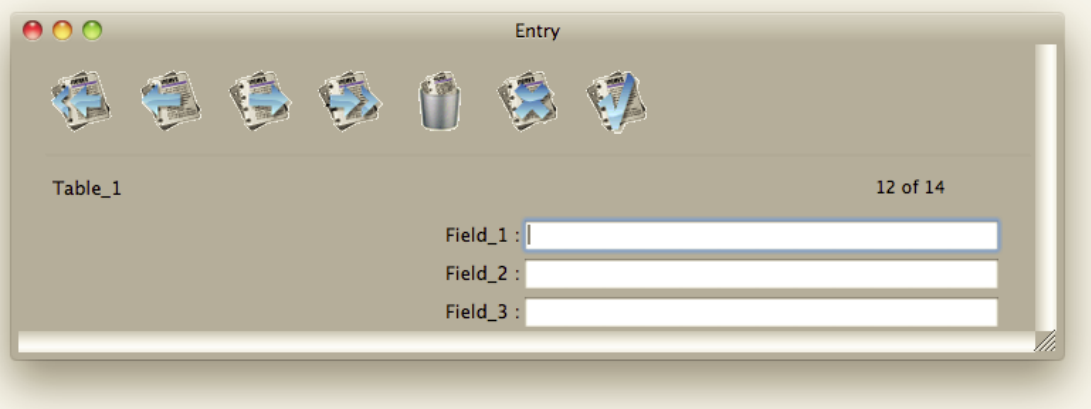
- You can only create a sheet window under Mac OS X if the last open window is visible and a document type (form).
- The command opens a type 1 (Modal dialog box) window instead of a type 33 window or type 8 (Plain) window instead of type 34:
  - if the last opened window is not visible or is not a document type,
  - under Windows.
- Since a sheet window must be drawn above a form, its display is pushed back in the On load event of the first form loaded in the window (see example 4 of the **Open window** command).
- Usage: **DIALOG**, **ADD RECORD** or equivalent, under Mac OS (not standard under Windows).

### Pop up window (32)

This type of window has the same basic characteristics of the Plain dialog box (2) type windows and features the following advanced specifics:

- The window is automatically closed and the "cancel" event is passed to the window when:
  - a click occurs outside the window;
  - the background window or the MDI (Multiple Document Interface) window is moved;
  - the user clicks the **Esc** key.
- This window is displayed in front of its "parent" window (it must not be used as the main window of the process). The background window is not disabled. However, it no longer receives events.
- You cannot resize or move the window using the mouse; however, when performing these actions programmatically, the redraw of background items is optimized.
- Usage: This type of window is primarily used to generate pop-up menus related to 3D “bevel” or “toolbar” type buttons.
- Limitations:
  - It is not possible to display pop-up menu objects inside this type of window.
  - Beginning with version 13 of 4D, this type of window does not permit the display of help tips under Mac OS.

### Texture appearance (2048)




Under Mac OS, it is possible to apply a texture appearance to windows. This type of look is found throughout the Macintosh interface. Under Windows, this property has no effect.

To apply a texture appearance to a window created by the **Open window** command, you can just add the Texture appearance constant to the window type set in the *type* parameter. For example:

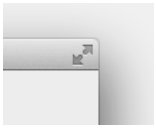
```
$win:=Open window(10;80;-1;-1;Plain window+Texture appearance;"")
```

This look can be associated with the following types of windows:

- Plain window
- Plain no zoom box window
- Plain fixed size window
- Movable dialog box
- Round corner window

### Has full screen mode Mac (65536)

The "full screen" option is available beginning with 4D v14 under OS X for document type windows. When this option is used, a "Full screen" button is displayed in the top right corner of the window:



When the user clicks on this icon, the window switches to full screen and 4D automatically hides the main tool bar.
To use this option, you just add the Has full screen mode Mac constant to the *type* parameter for the **Open window**, **Open form window** and **_o_Open external window** commands. For example, this code creates a form window with a full-screen button under OS X:

```
$win:=Open form window([Interface];"User_Choice";Plain form window+Form has full screen mode Mac)
DIALOG([Interface];"User_Choice")
```

**Note:** Under Windows, this option has no effect.

# CLOSE WINDOW

CLOSE WINDOW {{ window }}

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number, or Frontmost window of current process, if omitted |

## Description

**CLOSE WINDOW** closes the active window opened by the **Open window** or **Open form window** command in the current process. **CLOSE WINDOW** has no effect if a custom window is not open; it does not close system windows. **CLOSE WINDOW** also has no effect if called while a form is active in the window. You must call **CLOSE WINDOW** when you are done using a window opened by **Open window** or **Open form window**.

It is useless to pass a number to **CLOSE WINDOW** when closing a window previously opened by the **Open window** or **Open form window** function, since a call to **CLOSE WINDOW** will always close the last window created by one of these commands.

If you pass an external window reference number in the *Window* parameter, **CLOSE WINDOW** closes the specified external window. For more information about external windows, refer to the **_o_Open external window** function.

## Example

The following example opens a form window and adds new records with the **ADD RECORD** command. When the records have been added, the window is closed with **CLOSE WINDOW**:

```
FORM SET INPUT([Employees];"Entry")
$winRef:=Open form window([Employees];"Entry")
Repeat
   ADD RECORD([Employees]) //Add a new employee record
Until(OK=0) //Loop until the user cancels
CLOSE WINDOW //Close the window
```

# CONVERT COORDINATES

CONVERT COORDINATES ( xCoord ; yCoord ; from ; to )

| Parameter | Type | | Description |
|---|---|---|---|
| xCoord | Longint variable | ⇒ | Horizontal coordinate of a point (initial) |
| | | ⇐ | Horizontal coordinate of a point (converted) |
| yCoord | Longint variable | ⇒ | Vertical coordinate of a point (initial) |
| | | ⇐ | Vertical coordinate of a point (converted) |
| from | Longint | ⇒ | Coordinates system to convert from |
| to | Longint | ⇒ | Coordinates system to convert to |

## Description

The **CONVERT COORDINATES** command converts the (x;y) coordinates of a point from one coordinate system to another. The input and output coordinate systems supported are forms (and subforms), windows, and the screen. For example, you can use this command to get the coordinates in the main form of an object belonging to a subform. This makes it easy to create a context menu at any custom position.

In *xCoord* and *yCoord*, pass as variables the (x;y) coordinates of the point you want to convert. After the command is executed, these variables will contain the converted values.

In the *from* parameter, pass the initial coordinate system the input point is using, and in the *to* parameter, pass the coordinate system into which it must be converted. Both parameters can take one of the following constant values, added to the "**Windows**" theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| XY Current form | Longint | 1 | Origin is top left corner of current form |
| XY Current window | Longint | 2 | Origin is top left corner of current window |
| XY Main window | Longint | 4 | On Windows: origin is top left corner of main window; on OS X: same as XY Screen |
| XY Screen | Longint | 3 | Origin is top left corner of main screen (same as for **SCREEN COORDINATES** command) |

When this command is called from the method of a subform or a subform's object, and if one of the selectors is XY Current form, then the coordinates are relative to the subform itself, not to its parent form.

When converting from/to the position of a form window (for example when converting from the results of **GET WINDOW RECT**, or to values passed to **Open form window**), XY Main window must be used since it is the coordinate system used by window commands on Windows. It can also be used for this purpose on OS X, where it is equivalent to XY Screen.

When *from* is XY Current form and the point is in the body section of a list form, the result depends on the calling context of the command:

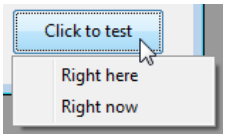
- If the command is called in the On Display Detail event, the resulting point is located in the display of the record being drawn on screen.
- If the command is called outside of an On Display Detail event but while a record is being edited, the resulting point is located in the display of the record being edited.
- Otherwise, the resulting point is located in the display of the first record.

## Example 1

You want to open a pop-up menu at the bottom left corner of the "MyObject" object.

```
// OBJECT GET COORDINATES works in the current form coordinate system
// Dynamic pop-up menu uses the current window coordinate system
// We need to convert the values
C_LONGINT($left;$top;$right;$bottom)
C_TEXT($menu)
OBJECT GET COORDINATES(*;"MyObject";$left;$top;$right;$bottom)
CONVERT COORDINATES($left;$bottom;XY Current form;XY Current window)
$menu:=Create menu
APPEND MENU ITEM($menu;"Right here")
```

```
APPEND MENU ITEM($menu;"Right now")
Dynamic pop up menu($menu;"";$left;$bottom)
RELEASE MENU($menu)
```




## Example 2

You want to open a pop-up window at the position of the mouse cursor. On Windows, you need to convert the coordinates since **GET MOUSE** (with the * parameter) returns values based on the position of the MDI window:

```
C_LONGINT($mouseX;$mouseY;$mouseButtons)
C_LONGINT($window)
GET MOUSE($mouseX;$mouseY;$mouseButtons)
CONVERT COORDINATES($mouseX;$mouseY;XY Current window;XY Main window)
$window:=Open form window("PopupWindowForm";Pop up form window;$mouseX;$mouseY)
DIALOG("PopupWindowForm")
CLOSE WINDOW($window)
```

# Current form window

Current form window -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | WinRef | | Current form window reference number |

## Description

The **Current form window** command returns the reference of the current form window. If no window has been set for the current form, the command returns 0.

The current form window can be generated automatically using a command such as **ADD RECORD**, following a user action or by using the **Open window** or **Open form window** commands.

# DRAG WINDOW

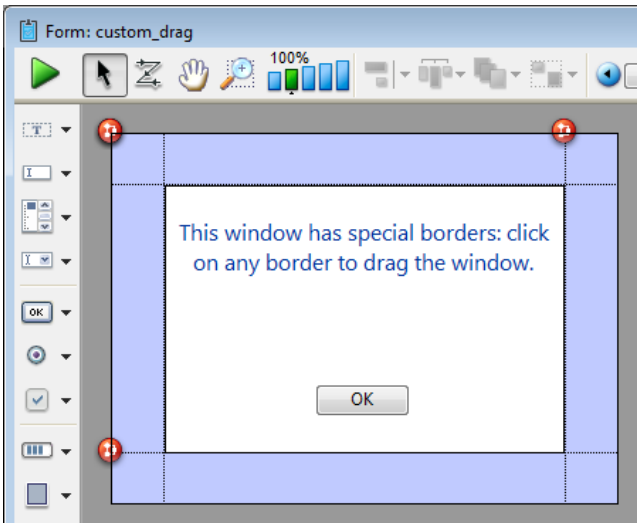
DRAG WINDOW
Does not require any parameters

## Description

The **DRAG WINDOW** command lets users drag the window on which they clicked following the movements of the mouse. Usually you call this command from within an object method of an object that can respond instantaneously to mouse clicks (i.e., invisible buttons).

## Example

The following form, shown here in the Form editor, contains a colored frame, above which are four invisible buttons for each side:



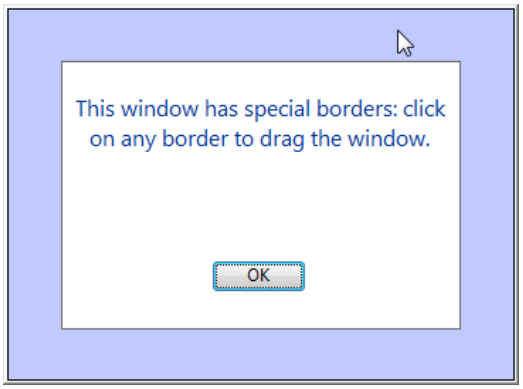

Each button has the following method:

```
DRAG WINDOW //Start dragging window when clicked
```

After executing the following project method:

```
$winRef:=Open form window("custom_drag";Modal form dialog box)
DIALOG("custom_drag")
CLOSE WINDOW
```

You obtain a window similar to this:




Then you can drag the window by clicking anywhere on the borders.

# ERASE WINDOW

ERASE WINDOW {( window )}

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇨ | Window reference number, or Frontmost window of current process, if omitted |

## Description

The **ERASE WINDOW** command clears the contents of the window whose reference number is passed in *window*.

If you omit the *window* parameter, **ERASE WINDOW** clears the contents of the frontmost window for the current process.

Usually, you will use **ERASE WINDOW** in combination with **MESSAGE** and **GOTO XY**. In this case, **ERASE WINDOW** clears the contents of the window and moves the cursor to the upper-left corner of the window, the **GOTO XY**(0; 0) position.

Do not confuse **ERASE WINDOW**, which clears the contents of a window, with **CLOSE WINDOW**, which removes the window from the screen.

# Find window

Find window ( left ; top {; windowPart} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| left | Longint | ⇒ | Global left coordinate |
| top | Longint | ⇒ | Global top coordinate |
| windowPart | Longint | ⇐ | 3 if window is "touched", 0 otherwise |
| Function result | WinRef | ⇔ | Window reference number |

## Description

The **Find window** command returns (if any) the reference number of the first window “touched” by the point whose coordinates passed in *left* and *top*.

The coordinates must be expressed relative to the top left corner of the contents area of the application window (Windows) or to the main screen (Macintosh).

The *windowPart* parameter returns 3 if the window is touched, and 0 otherwise. (**Compatibility note:** Starting with 4D v14, the constants of the **Find Window** theme are obsolete).

# Frontmost window

Frontmost window {( * )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| * | Operator | ⇒ | If specified, take floating windows into account If omitted, ignore floating windows |
| Function result | WinRef | ⇐ | Window reference number |

## Description

The **Frontmost window** command returns the window reference number of the frontmost window.

# GET WINDOW RECT

GET WINDOW RECT ( left ; top ; right ; bottom {; window} )

| Parameter | Type | | Description |
|---|---|---|---|
| left | Longint | ⇦ | Left coordinate of window's contents area |
| top | Longint | ⇦ | Top coordinate of window's contents area |
| right | Longint | ⇦ | Right coordinate of window's contents area |
| bottom | Longint | ⇦ | Bottom coordinate of window's contents area |
| window | WinRef | ⇨ | Window reference number, or Frontmost window of current process, if omitted or MDI window if -1 (Windows) |

## Description

The **GET WINDOW RECT** command returns the coordinates of the window whose reference number is passed in *window*. If the window does not exist, the variable parameters are left unchanged.

If you omit the *window* parameter, **GET WINDOW RECT** applies to the frontmost window for the current process.

The coordinates are expressed relative to the top left corner of the contents area of the application window (on Windows) or of the main screen (on Macintosh). The coordinates return the rectangle corresponding to the contents area of the window (excluding title bars and borders).

**Note:** Under Windows, if you pass -1 in *window*, **GET WINDOW RECT** returns the coordinates of the application window (MDI window). These coordinates correspond to the contents area of the window (excluding menu bars and borders).

## Example

See example for the **WINDOW LIST** command.

# Get window title

Get window title {( window )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number, or Frontmost window of current process, if omitted |
| Function result | String | ⇐ | Window title |

## Description

The **Get window title** command returns the title of the window whose reference number is passed in *window*. If the window does not exist, an empty string is returned.

If you omit the *window* parameter, **Get window title** returns the title of the frontmost window for the current process.

## Example

See example for the **SET WINDOW TITLE** command.

# HIDE TOOL BAR

HIDE TOOL BAR
Does not require any parameters

## Description

The **HIDE TOOL BAR** command handles the display of custom toolbars created by the **Open form window** command for the current process.

If a toolbar window has been created by the **Open form window** command with the Toolbar form window option, the command hides this window. If the toolbar window was already hidden or if no window of this type has been created, the command does nothing.

## Example

On OS X, you have defined a custom toolbar and a standard window that has the Has full screen mode Mac option. When a standard window is maximized by a user while the toolbar window is displayed, you do not want the toolbar to overlap the maximized window.

To prevent this, in the On Resize form event of the standard window, you need to detect when the window has entered full screen mode and then call **HIDE TOOL BAR**:

```
Case of
   :(Form event=On Resize)
      GET WINDOW RECT($left;$top;$right;$bottom)
      If(Screen height=($bottom-$top))
         HIDE TOOL BAR
      Else
         SHOW TOOL BAR
      End if
End case
```

# HIDE WINDOW

HIDE WINDOW {( window )}

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ➡ | Window reference number or Current process frontmost window, if omitted |

## Description

The **HIDE WINDOW** command hides the window whose number was passed in *window* or, if this parameter is omitted, the current process frontmost window. For example, this command lets you display only the active window in a process that consists of several processes.

The window disappears from the screen but remains open. You can still apply any changes supported by 4D windows programmatically.

To display a window that was previously hidden by the **HIDE WINDOW** command:

- Use the **SHOW WINDOW** command and pass the window reference ID.
- Use the **Process** page of the Runtime Explorer. Select the process in which the window is handled, then click on the **Show** button.

To hide all the windows of a process, use the **HIDE PROCESS** command.

## Example

This example corresponds to a method of a button located in an input form. This button opens a dialog box in a new window that belongs to the same process. In this example, the user wants to hide the other windows of the process (an entry form and a tool palette) while displaying the dialog box. Once the dialog box is validated, other process windows are displayed again.

```
  ` Object method for the "Information" button

HIDE WINDOW(Entry) ` Hide the entry window
HIDE WINDOW(Palette) ` Hide the palette
$Infos:=Open window(20;100;500;400;8) ` Create the information window
... ` Place here instructions that are dedicated to the dialog management
CLOSE WINDOW($Infos) ` Close the dialog
SHOW WINDOW(Entry)
SHOW WINDOW(Palette) ` Display the other windows
```

# MAXIMIZE WINDOW

MAXIMIZE WINDOW {{ window )}

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number or if omitted, all current process frontmost windows (Windows) or current process frontmost window (Mac OS) |

## Description

The **MAXIMIZE WINDOW** command triggers the expansion of the window whose reference number was passed in *window*. If this parameter is omitted, the effect is the same but is applied to all the frontmost windows of the current process (Windows) or to the frontmost window of the current process (Mac OS).

This command has the same effect as a click on the zoom box of a 4D application window. Windows you want to maximize must have a zoom box. If the *window* type does not have a zoom box, the command does nothing (for more information about this point, refer to **Window Types** section).

A later click on the zoom box or a call to the **MINIMIZE WINDOW** command reduces the window to its initial size. On Windows, a call to **MINIMIZE WINDOW** without parameters sets the size of all application windows to their initial sizes.

If *window* is already maximized, the command does nothing.

### On Windows

The size of the window is increased to match the current size of the application window. The maximized window is set to be the frontmost window. If you do not pass the *window* parameter, the command is applied to all the application windows.

Windows zoom box

In cases where the command is applied to a window whose size is subject to constraints (for example, a form window):

- If no size constraint is in conflict with the target size, the window is "maximized" (i.e., it is restored to the size of the parent MDI ("Multiple Document Interface") window; its title bar and borders are hidden, and its control buttons - minimize, restore and close - are moved to the right of the application menu bar).
- If at least one size constraint is in conflict (for example, if the width of the MDI window is 100 and the form window's maximum width is set to 80), the window is not "maximized", but only restored to its maximum allowed size. This size is defined either by the MDI window, or by the constraint. This way, the interface remains consistent when windows with constraints are resized.

### On Mac OS

The size of the window is increased to match the size of its contents. If you do not pass the *window* parameter, the command is applied to the frontmost window of the current process.

Zoom box on Mac OS

- The zoom is based on the contents of the window; so, the command must be called in a context where the contents of the window are defined, for example in a form method. Otherwise, the command does nothing.
- The window is set to its "maximum" size. If the window is actually a form whose size was defined in the form properties, the window size is set to those values.

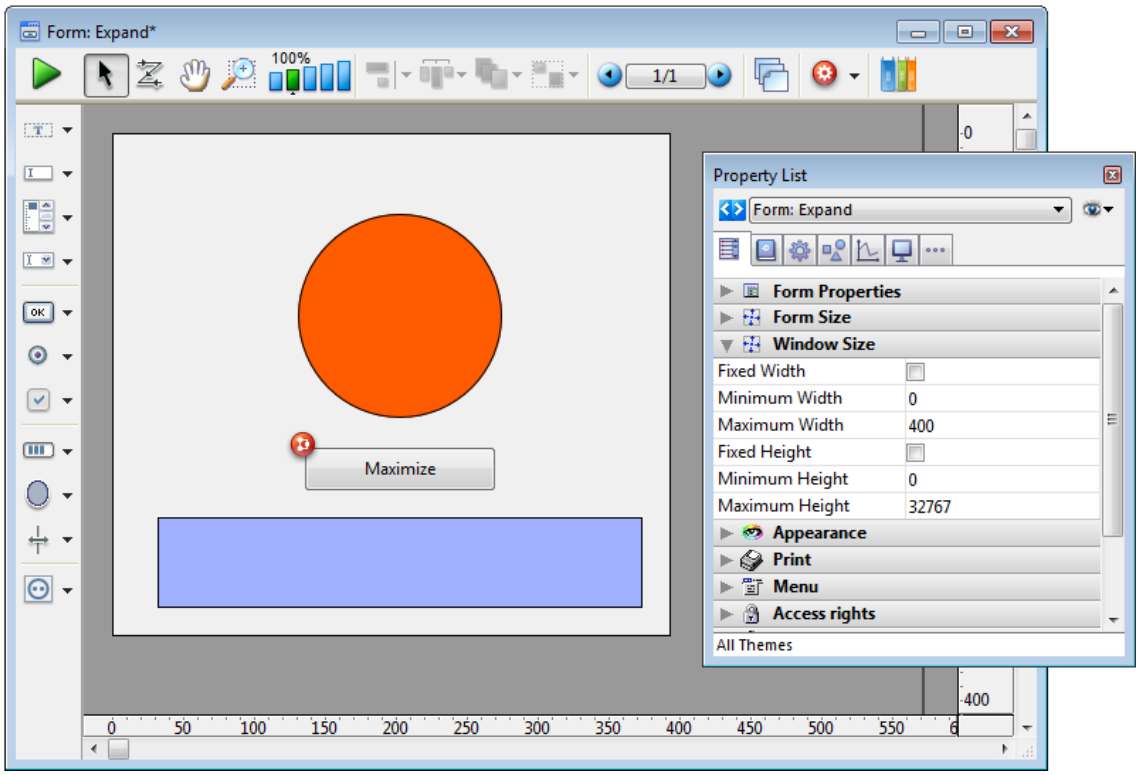
## Example 1

This example sets the window size of your form to full screen when it is opened. To achieve this, the following code is placed in the form method:

```
` In the Form method
MAXIMIZE WINDOW
```

## Example 2

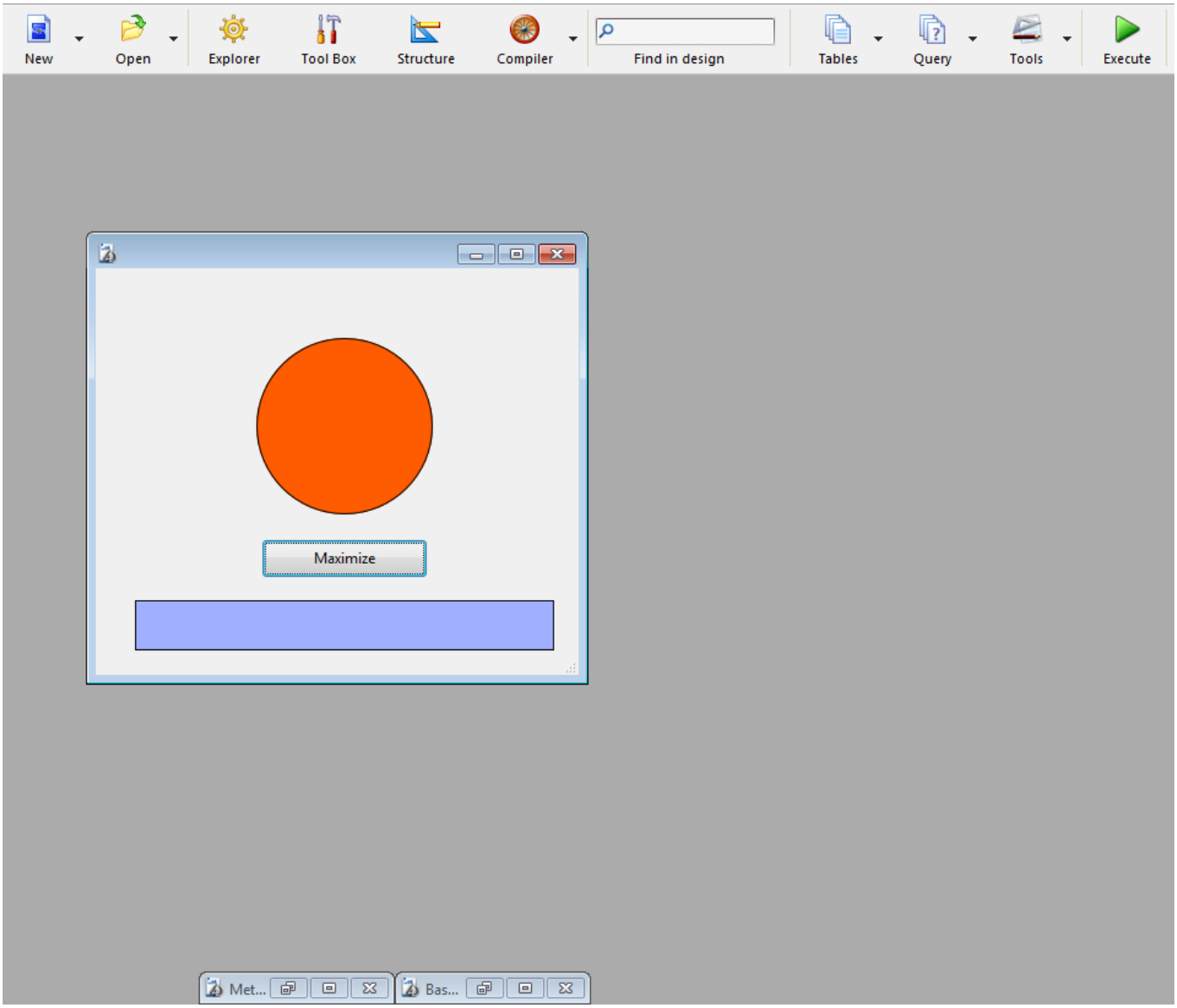
This example illustrates how size constraints are handled on Windows. The following form has a size constraint (maximum width=400):



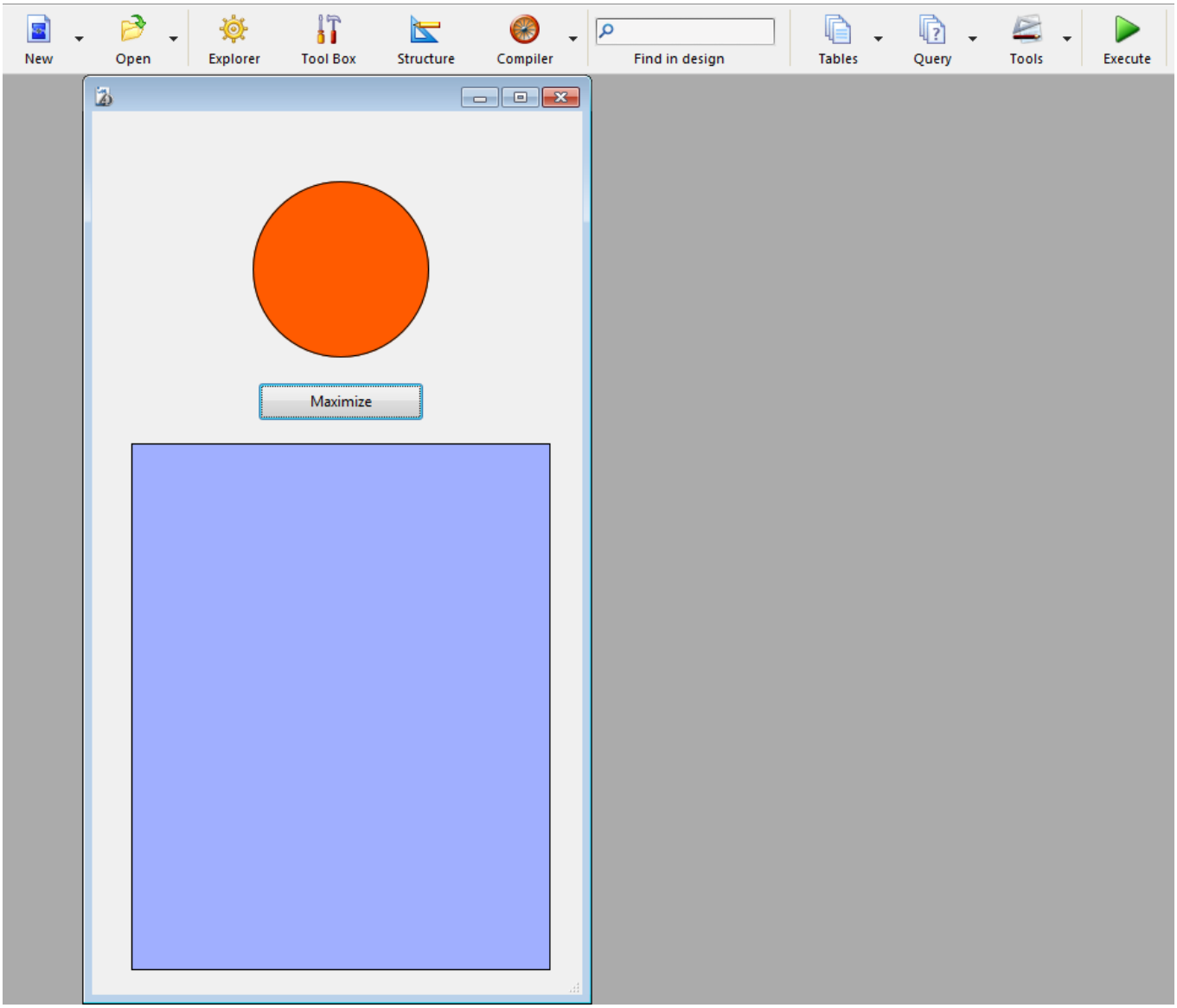

The method of the button contains simply:

```
MAXIMIZE WINDOW(Current form window)
```

In this context, when the user clicks on the button:




... the window is not "maximized"; only its height is increased:


New
Open
Explorer
Tool Box
Structure
Compiler
Find in design
Tables
Query
Tools
Execute
Maximize

# MINIMIZE WINDOW

MINIMIZE WINDOW {{ window }}

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number or if omitted, all the current process frontmost windows (Windows) or current process frontmost window (Mac OS) |

## Description

The **MINIMIZE WINDOW** command sets the size of the window whose number is passed as *window* to the size it was before being maximized. If *window* is omitted, the command applies to each window of the application (Windows) or to the frontmost window of the process (on Mac OS).

This command has the same effect as one click on the reduction box of the 4D application:

### On Windows

The size of the window is set to its initial size, i.e., its size before being maximized. If the *window* parameter is omitted, all the application windows are set to their initial sizes.

Reduction box on Windows

### On Mac OS

The size of the window is set to its initial size (i.e. its size before being maximized). If the window parameter is omitted, the frontmost window of the current process is set to its initial size.

Reduction/zoom box on Mac OS

If the windows to which the command is applied were not previously maximized (manually or using **MAXIMIZE WINDOW**), or if the window type does not include a zoom box, the command has no effect. For more information on window types, refer to the **Window Types** section.

**Note:** This function is not to be confused with minimizing a window to a button (Windows) or in the Dock (Mac OS), which is triggered by a click on the button shown:

Windows

Mac OS

# Next window

Next window ( window ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number |
| Function result | WinRef | ⇒ | Window reference number |

## Description

The **Next window** command returns the window reference number of the window “behind” the window you pass in *window* (based on the front-to-back order of the windows).

# Open form window

Open form window ( {aTable ;} formName {; type {; hPos {; vPos {; *}}}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| aTable | Table | ➡ | Table of the form or Default table, if omitted |
| formName | String | ➡ | Name of the form |
| type | Longint | ➡ | Window type |
| hPos | Longint | ➡ | Horizontal position of the window |
| vPos | Longint | ➡ | Vertical position of the window |
| * | Operator | ➡ | Save current position and size of the window |
| Function result | WinRef | ⬅ | Window reference number |

## Description

The **Open form window** command opens a new window using the size and resizing properties of the form *formName*.

Note that *formName* is not displayed in the window. If you want to display the form, you have to call a command which loads a form (**ADD RECORD** for example).

By default (if the *type* parameter is not passed), a standard window with a close box is opened. Unlike the **Open window** command, no method is associated to the window's close box. Clicking on this close box cancels and closes the window, except if the On Close Box form event has been activated for the form. In this case, the code associated with the On Close Box event will be executed.

If *formName* is resizable, the window opened will contain a zoom box as well as a grow box.

**Note:** To know the main properties of a form, use the **FORM GET PROPERTIES** command.

The optional *type* parameter allows you to specify a type for the window. You must pass one of the following predefined constants (placed in the **Open Form Window** theme):

| Constant | Type | Value |
|---|---|---|
| Form has full screen mode Mac | Longint | 65536 |
| Modal form dialog box | Longint | 1 |
| Movable form dialog box | Longint | 5 |
| Palette form window | Longint | 1984 |
| Plain form window | Longint | 8 |
| Pop up form window | Longint | 32 |
| Sheet form window | Longint | 33 |
| Toolbar form window | Longint | 35 |

**Notes:**

- The attributes (grow box, close box...) of the window created depend on the interface specifications of the operating system for the chosen *type*. It is therefore possible to obtain different results depending on the platform used.
- The Form has full screen mode Mac constant must be added to one of the other type constants.
- For more information about window types, refer to the **Window Types** section. Note that only the types listed in the **Open Form Window** theme can be used with the **Open form window** command.

When the Toolbar form window constant is passed, the window is created with the location, size and graphical properties of a toolbar, i.e.:

- The window will always be displayed just under the menu bar.
- The window's horizontal size will be automatically adjusted to fill all available horizontal space on the desktop (on OS X) or inside 4D's main window (on Windows). The window's vertical size is based on the form properties like all other form window types.
- The window has no border, cannot be moved and cannot be resized manually, and *hPos*, *vPos* and * parameters are ignored if present.
- It is not possible to create two different toolbar windows at the same time. If **Open form window** is called with the Toolbar form window *type* while a toolbar window already exists, an error -10613 ("Cannot create two form windows of type toolbar") is generated.

**Toolbar form windows and OS X full screen mode:** If your application displays both a toolbar window and standard windows that support full screen mode (Form has full screen mode Mac option), interface rules require that you hide the toolbar when a standard window enters full screen mode. To know if a window has switched to full screen mode, just test whether its vertical size is exactly the same as the screen's height (see the **HIDE TOOL BAR** command).

The optional parameter *hPos* allows you to define the horizontal position of the window. You can pass a defined position, expressed in points, to this parameter (refer to the **Open window** command) or one of the following predefined constants placed in the **Open Form Window** theme:

| Constant | Type | Value |
|---|---|---|
| Horizontally centered | Longint | 65536 |
| On the left | Longint | 131072 |
| On the right | Longint | 196608 |

The optional parameter *vPos* allows you to define the vertical position of the window. You can pass a defined position, expressed in points, to this parameter (refer to the **Open window** command) or one of the following predefined constants placed in the **Open Form Window** theme:

| Constant | Type | Value |
|---|---|---|
| At the bottom | Longint | 393216 |
| At the top | Longint | 327680 |
| Vertically centered | Longint | 262144 |

These parameters take into account the presence of the tool bar and menu bar as well as the current size of the application's window (on Windows).

If you pass the optional parameter *, the current position and size of the window are memorized when closed. When the window is reopened again, its previous position and size are respected. In this case, the *vPos* and *hPos* parameters are only used the first time the window is opened.

**Note:** To reopen a window at *vPos* and *hPos* initial coordinates when the * parameter is used, hold down the **Shift** key while the window is being opened.

## Example 1

The following statement opens a standard window with a close box and automatically adjusts it to be the same size as the "Input" form. Since the form has been defined as resizable, the window also has a grow and a zoom box:

```
$winRef :=Open form window([Table1];"Enter")
```

## Example 2

The following statement opens a floating palette in the upper left portion of the screen based on a project form named "Tools". This palette uses the last position it was in when the user closed it each time it is reopened:

```
$winRef :=Open form window("Tools";Palette form window;On the left;At the top;*)
```

# Open window

Open window ( left ; top ; right ; bottom {; type {; title {; controlMenuBox}}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| left | Longint | ⇒ | Global left coordinate of window contents area |
| top | Longint | ⇒ | Global top coordinate of window contents area |
| right | Longint | ⇒ | Global right coordinate of window contents area, or -1 for using form default size |
| bottom | Longint | ⇒ | Global bottom coordinate of window contents area, or -1 for using form default size |
| type | Longint | ⇒ | Window type |
| title | String | ⇒ | Title of window or "" for using default form title |
| controlMenuBox | String | ⇒ | Method to call when the Control-menu box is double-clicked or the Close box is clicked |
| Function result | WinRef | ⇐ | Window reference number |

## Description

**Open window** opens a new window with the dimensions given by the first four parameters:

- *left* is the distance in pixels from the left edge of the application window to the left internal edge of the window.
- *top* is the distance in pixels from the top of the application window to the top internal edge of the window.
- *right* is the distance in pixels from the left edge of the application window to the right internal edge of the window.
- *bottom* is the distance in pixels from the top of the application window to the bottom internal edge of the window.

**Compatibility note: Open window** integrates various options which have evolved over the versions, and is now only kept for compatibility reasons. When you write new code for managing windows, we strongly recommended using the **Open form window** command, which is better suited to current interfaces.

If you pass -1 in both *right* and *bottom,* you instruct 4D to automatically size the window under the following conditions:

- You have designed a form and set its Sizing Options in the Design environment Form properties window
- Before calling **Open window**, you selected the form using the **FORM SET INPUT** command, to which you passed the optional * parameter.

**Important:** This automatic sizing of the window will occur only if you made a prior call to **FORM SET INPUT** for the form to be displayed, and if you passed the * optional parameter to **FORM SET INPUT**.

- The *type* parameter is optional. It represents the type of window you want to display, and corresponds to the different windows shown in the section **Window Types** (constants of the **Open Window** theme). If the window type is negative, the window created is a floating window. If the type is not specified, type 1 is used by default.
- The *title* parameter is the optional title for the window

If you pass an empty string ("") in *title,* you instruct 4D to use the Window Title set in the Design environment Form Properties window for the form to be displayed.

**Important:** The default form title will be set to the window only if you made a prior call to **FORM SET INPUT** for the form to be displayed, and if you passed the * optional parameter to **FORM SET INPUT**.

- The *controlMenuBox* parameter is the optional Control-menu box method for the window. If this parameter is specified, a Control-menu box (Windows) or a Close Box (Macintosh) is added to the window. When the user double-clicks the Control-menu box (Windows) or clicks on the Close Box (Macintosh), the method passed in *controlMenuBox* is called.

**Note:** You can also manage the closing of the window from within the form method of the form displayed in the window when an On Close Box event occurs. For more information, see the command **Form event**.

If more than one window is open for a process, the last window opened is the active (frontmost) window for that process. Only information within the active window can be modified. Any other windows can be viewed. When the user types, the active window will always come to the front, if it is not already there.

Forms are displayed inside an open window. Text from the **MESSAGE** command also appears in the window.

**Open window** returns a *WinRef* type window reference, which can be used by window management commands (see the "**WinRef**" section).

## Example 1

The following project method opens a window centered in the main window (Windows) or in the main screen (Macintosh). Note that it can accept two, three, or four parameters:

```
  ` OPEN CENTERED WINDOW project method
  ` $1 - Window width
  ` $2 - Window height
  ` $3 - Window type (optional)
  ` $4 - Window title (optional)
$SW:=Screen width¥2
$SH:=(Screen height¥2)
$WW:=$1¥2
$WH:=$2¥2
Case of
   :(Count parameters=2)
      Open window($SW-$WW;$SH-$WH;$SW+$WW;$SH+$WH)
   :(Count parameters=3)
      Open window($SW-$WW;$SH-$WH;$SW+$WW;$SH+$WH;$3)
   :(Count parameters=4)
      Open window($SW-$WW;$SH-$WH;$SW+$WW;$SH+$WH;$3;$4)
End case
```

After the project method is written, you can use it this way:

```
OPEN CENTERED WINDOW(400;250;Movable dialog box;"Update Archives")
DIALOG([Utility Table];"UPDATE OPTIONS")
CLOSE WINDOW
If(OK=1)
  ` ...
End if
```

## Example 2

The following example opens a floating window that has a Control-menu box (Windows) or Close Box (Macintosh) method. The window is opened in the upper right hand corner of the application window.

```
$myWindow:=Open window(Screen width-149;33;Screen width-4;178;-Palette window;"";"CloseColorPalette")
DIALOG([Dialogs];"Color Palette")
```

The ***CloseColorPalette*** method calls the **CANCEL** command:

```
CANCEL
```

## Example 3

The following example opens a window whose size and title come from the properties of the form displayed in the window:

```
FORM SET INPUT([Customers];"Add Records";*)
$myWindow:=Open window(10;80;-1;-1;Plain window;"")
Repeat
   ADD RECORD([Customers])
Until(OK=0)
```

**Reminder:** In order to have **Open window** automatically use the properties of the form, you must call **FORM SET INPUT** with the optional * parameter, and the properties of the form must have been set accordingly in the Design environment.

## Example 4

This example illustrates the “delay” mechanism for displaying sheet windows under Mac OS X:

```
$myWindow:=Open window(10;10;400;400;Sheet window)
  `For the moment, the window is created but remains hidden
DIALOG([Table];"dialForm")
  `The On Load event is generated then the sheet window is displayed: it "drops down" from the bottom
  `of the title bar
```

# REDRAW WINDOW

REDRAW WINDOW {( window )}

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number, or Frontmost window of current process, if omitted |

## Description

The **REDRAW WINDOW** command provokes a graphical update of the window whose reference number you pass in *window*.

If you omit the *window* parameter, **REDRAW WINDOW** applies to the frontmost window for the current process.

**Note:** 4D handles the graphical updates of the windows each time you move a window, resize it, or bring it to the front, as well as when you change the form and/or the values displayed in the window. You will rarely use this command.

# RESIZE FORM WINDOW

RESIZE FORM WINDOW ( width ; height )

| Parameter | Type | | Description |
|---|---|---|---|
| width | Longint | ⇒ | Pixels to add to or remove from the current form window width |
| height | Longint | ⇒ | Pixels to add to or remove from the current form window height |

## Description

The **RESIZE FORM WINDOW** command lets you change the size of the current form window.

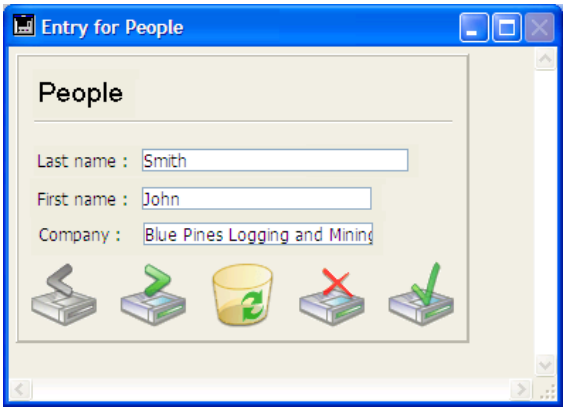
Pass the number of pixels that you would like to add to the current window size in the *width* and *height* parameters. Pass 0 in either parameter if you do not wish to change the current size. To reduce the size, pass a negative value in the *width* and *height* parameters.

This command produces the exact same result as a manual window resize using the resize box (if the window type allows it). As a result, the command takes into consideration resize properties for objects and size limitations defined in the form properties. If, for example, the command resizes a window to a size greater than what is allowed in the form, the command will have no effect.

Please note that this behavior is different than that of the **SET WINDOW RECT** command, which does not take form properties nor content into account when resizing the window. Also, note that this command does not necessarily modify the form size. To modify the size of a form by programming, please see the **FORM SET SIZE** command.

## Example

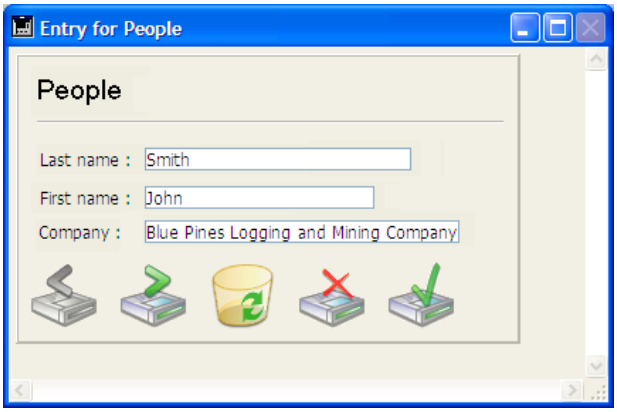
Given the following window (the fields and frame have the “Grow” property for horizontal resizing):




After execution of this line:

```
RESIZE FORM WINDOW(25;0)
```

... the window appears as follows:



# SET WINDOW RECT

SET WINDOW RECT ( left ; top ; right ; bottom {; window}{; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| left | Longint | ⇒ | Global left coordinate of window's contents area |
| top | Longint | ⇒ | Global top coordinate of window's contents area |
| right | Longint | ⇒ | Global right coordinate of window's contents area |
| bottom | Longint | ⇒ | Global bottom coordinate of window's contents area |
| window | WinRef | ⇒ | Window reference number, or Frontmost window of current process, if omitted |
| * | Operator | ⇒ | If omitted (default) = change window to foreground<br>If passed = do not change the level of the window |

## Description

The **SET WINDOW RECT** command changes the global coordinates of the the window whose reference number is passed in *window*. If the window does not exist, the command does nothing.

If you omit the *window* parameter, **SET WINDOW RECT** applies to the frontmost window for the current process.

This command can resize and move the window, depending on the new coordinates passed.

The coordinates must be expressed relative to the top left corner of the contents area of the application window (on Windows) or to the main screen (on Macintosh). The coordinates indicate the rectangle corresponding to the contents area of the window (excluding title bars and borders).

**Warning:** Be aware that by using this command, you may move a window beyond the limits of the main window (on Windows) or of the screens (on Macintosh). To prevent this, use commands such as **Screen width** and **Screen height** to double-check the new coordinates of the window.

By default, executing this command automatically moves the window designated by the *window* parameter to the foreground (if this parameter is used). You can disable this by passing the * as the last parameter. In this case, the command no longer changes the original level ("z" coordinate) of the window.

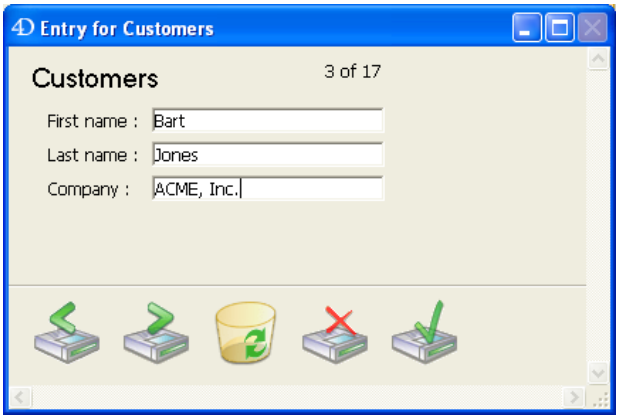
This command does not affect form objects. If the window contains a form, the form objects are not moved or resized by the command (regardless of their properties). Only the window is modified. In order to modify a form window while taking the resizing properties and the objects it contains into account, you must use the **RESIZE FORM WINDOW** command.

## Example 1

See example for the **WINDOW LIST** command.

## Example 2

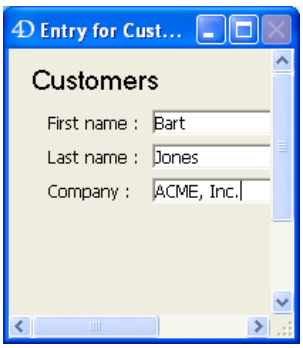
Given the following window:




After execution of the following line:

```
SET WINDOW RECT(100;100;300;300)
```

The window appears as follows:


Entry for Cust...
Customers
First name :
Bart
Last name :
Jones
Company :
ACME, Inc.

# SET WINDOW TITLE

SET WINDOW TITLE ( title {; window} )

| Parameter | Type | | Description |
|---|---|---|---|
| title | String | ⇒ | Window title |
| window | WinRef | ⇒ | Window reference number, or Frontmost window of current process, if omitted |

## Description

The **SET WINDOW TITLE** command changes the title of the window whose reference number is passed in *window* to the text passed in *title* (max. length 80 characters).

If the window does not exist, **SET WINDOW TITLE** does nothing.

If you omit the *window* parameter, **SET WINDOW TITLE** changes the title of the frontmost window for the current process.

**Note:** In the Design environment, 4D changes the window titles automatically —i.e., “Entry for Table” when you perform data entry. If you change a window title, 4D will probably override it. On the other hand, in the Application environment, 4D does not change the titles of the windows.

## Example

While performing data entry in a form, you click on a button that executes a lengthy operation (i.e., browsing programmatically related records shown in a subform). You keep informed about the progress of the operation using the title of the current window:

```
  ` bAnalysis button Object Method
Case of
   :(Form event=On Clicked)
  ` Save current window title in a local variable
      $vsCurTitle:=Get window title
  ` Start the lengthy operation
      FIRST RECORD([Invoice Line Items])
      For($vlRecord;1;Records in selection([Invoice Line Items]))
         DO SOMETHING
  ` Show progress information
         SET WINDOW TITLE("Processing Line Item #"+String($vlRecord))
      End for
  ` Restore original window title
      SET WINDOW TITLE($vsCurTitle)
End case
```

# SHOW TOOL BAR

SHOW TOOL BAR
Does not require any parameters

## Description

The **SHOW TOOL BAR** command handles the display of custom toolbars created by the **Open form window** command for the current process.

If a toolbar window has been created by the **Open form window** command with the Toolbar form window option, the command makes this window visible. If the toolbar window was already visible or if no window of this type has been created, the command does nothing.

## Example

Refer to the example of the **HIDE TOOL BAR** command.

# SHOW WINDOW

SHOW WINDOW {( window )}

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number or Current process frontmost window, if omitted |

## Description

The **SHOW WINDOW** command displays the window whose number was passed in *window*. If this parameter is omitted, the frontmost window of the current process will be displayed.

In order to use the **SHOW WINDOW** command, the window must have been hidden by using the **HIDE WINDOW** command. If the window is already displayed, the command does nothing.

## Example

Refer to the example of the **HIDE WINDOW** command.

# Tool bar height

Tool bar height -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| Function result | Longint | ➲ | Height (expressed in pixels) of tool bar or 0 if tool bar is hidden |

## Description

The **Tool bar height** command returns the height of the current visible tool bar, expressed in pixels. Depending on the context, it can be either the 4D Design mode toolbar, or a custom toolbar created with **Open form window** (the Design mode toolbar is automatically hidden when a custom toolbar is displayed).

If no tool bar is displayed, the command returns 0.

# Window kind

Window kind {( window )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number, or Frontmost window of current process, if omitted |
| Function result | Longint | ⇐ | Type of window |

## Description

The **Window kind** command returns the 4D type of the window whose reference number is passed in *window*. If the window does not exist, **Window kind** returns *0* (zero).

Otherwise, **Window kind** may return one of the following predefined constants found in the **Windows** theme:

| Constant | Type | Value |
|---|---|---|
| External window | Longint | 5 |
| Floating window | Longint | 14 |
| Modal dialog | Longint | 9 |
| Regular window | Longint | 8 |

If you omit the *window* parameter, **Window kind** returns the type of the frontmost window for the current process.

## Example

See example for the **WINDOW LIST** command.

# WINDOW LIST

WINDOW LIST ( windows {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| windows | Array | ⇐ | Array of window reference numbers |
| * | Operator | ⇒ | If specified, take floating windows into account If omitted, ignore floating windows |

## Description

The **WINDOW LIST** command populates the array *windows* with the window reference numbers of the windows currently open in all running processes (kernel or user processes).

If you do not pass the optional * parameter, floating windows are ignored.

## Example

The following project method tiles all the current open window, except floating windows and dialog boxes:

```
   ` TILE WINDOWS project method

WINDOW LIST($alWnd)
$vlLeft:=10
$vlTop:=80 ` Leave enough room for the Tool bar
For($vlWnd;1;Size of array($alWnd))
   If(Window kind($alWnd{$vlWnd})#Modal dialog)
      GET WINDOW RECT($vlWL;$vlWT;$vlWR;$vlWB;$alWnd{$vlWnd})
      $vlWR:=$vlLeft+($vlWR-$vlWL)
      $vlWB:=$vlTop+($vlWB-$vlWT)
      $vlWL:=$vlLeft
      $vlWT:=$vlTop
      SET WINDOW RECT($vlWL;$vlWT;$vlWR;$vlWB;$alWnd{$vlWnd})
      $vlLeft:=$vlLeft+10
      $vlTop:=$vlTop+25
   End if
End for
```

**Note:** This method could be improved by adding tests on the size of the main window (on Windows) or the size and location of the screens (on Macintosh).

# Window process

Window process {( window )} -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| window | WinRef | ⇒ | Window reference number |
| Function result | Longint | ⇐ | Process reference number |

## Description

The **Window process** command returns the process number that runs the window whose reference number is passed in *window*. If the window does not exist, 0 (zero) is returned.

If you omit the *window* parameter, **Window process** returns the process of the current frontmost window.

# _o_Open external window

_o_Open external window ( left ; top ; right ; bottom ; type ; title ; plugInArea ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| left | Longint | ⇒ | Global left coordinate of window contents area |
| top | Longint | ⇒ | Global top coordinate of window contents area |
| right | Longint | ⇒ | Global right coordinate of window contents area |
| bottom | Longint | ⇒ | Global bottom coordinate of window contents area |
| type | Longint | ⇒ | Window type |
| title | String | ⇒ | Title of window |
| plugInArea | String | ⇒ | External area command |
| Function result | WinRef | ⇐ | Reference number for window and external area |

## Compatibility note

The **_o_Open external window** command is declared obsolete in 4D beginning with version 16 and is only kept for compatibility reasons. Note that it is not supported in 64-bit versions of 4D.

# XML

- Overview of XML Utilities Commands
- XML DECODE
- XML GET ERROR
- XML GET OPTIONS
- XML SET OPTIONS
- *_o_XSLT APPLY TRANSFORMATION*
- *_o_XSLT GET ERROR*
- *_o_XSLT SET PARAMETER*

# Overview of XML Utilities Commands

This theme groups together the generic XML "utilities" commands of 4D. These are option- and error-management commands as well as commands specialized in XSL.

For general information about XML (overview, glossary) as well as the differences between DOM and SAX modes, please refer to the **Overview of XML DOM Commands** section.

## What is SVG?

SVG (Scalable Vector Graphics) is a file format used to describe vector graphics (extension .svg) in XML. The most common use of SVG is the publication of statistical or mapping data.

These files can be viewed in Web browsers either natively, or via plug-ins. 4D v11 includes an SVG rendering engine that lets you view SVG files in picture fields or variables. The **SVG EXPORT TO PICTURE** command can be used to generate a picture in 4D based on an SVG description. Also note that the **GRAPH** command can be used to take advantage of the integrated SVG engine of 4D.

For more information about this format, please refer to the following address: http://www.w3.org/Graphics/SVG/.

## About XSLT commands

Starting with 4D v14 R4, XSL transformation commands are declared obsolete and prefixed accordingly:

| Former name | 4D v14 R4 and higher |
|---|---|
| XSLT APPLY TRANSFORMATION | **_o_XSLT APPLY TRANSFORMATION** |
| XSLT GET ERROR | **_o_XSLT GET ERROR** |
| XSLT SET PARAMETER | **_o_XSLT SET PARAMETER** |

For compatibility, XSL transformations are still supported by 4D but their use is no longer recommended. In future versions of 4D, it will no longer be possible to use XSLT technology.

**Note for 4D Server 64-bit version for OS X:** XSLT is not included in the 64-bit version of 4D Server for OS X. As a result, executing one of these commands from this application generates an error 33, "Unimplemented command or function".

To replace XSLT technology in your databases, 4D provides two solutions:

- Using equivalent PHP functions available in the *libxslt* module, which is installed in 4D beginning with version 14.2. 4D provides a specific document to help you use PHP's XSL to replace 4D's XSLT commands: Download the "XSLT with PHP" document (PDF)
- Using the means provided by the **PROCESS 4D TAGS** command, whose capabilities have been expanded considerably starting with 4D v14 R4.

# XML DECODE

XML DECODE ( xmlValue ; 4Dobject )

| Parameter | Type | | Description |
|---|---|---|---|
| xmlValue | Text | ⇒ | Text type value coming from an XML structure |
| 4Dobject | Field, Variable | ⇐ | 4D variable or field receiving the converted XML value |

## Description

The **XML DECODE** command converts a value stored as an XML string into a 4D typed value. The conversion is carried out automatically according to the following rules:

| Value | Examples | Conversion on English system |
|---|---|---|
| number | <Price>8,5</Price><Price>8.5</Price> | Real: 8.5 |
| Boolean | <Double>1</Double> <Double>0</Double> or <Double>true</Double> <Double>false</Double> | Boolean: True/False |
| BLOB | | Base64 decoding |
| Picture | | Base64 decoding + BLOB to picture command |
| Dates | 2009-10-25T01:03:20+01:00 | Deletion of time part as well as time zone: !10/25/2009! |
| Time | 2009-10-25T01:03:20+01:00 | Deletion of date part as well as time zone: ?01:03:20? |

## Example

Importing data from an XML document in which values are stored as attributes.
Example of XML document:

```
<CD Date="2003-01-01T00:00:00Z"
Description="This double CD reissued by EMI in 1995 combines 4 Stabat mater hymns. One by Rossini interpreted by the Berlin Symphony Orchestra, directed by Karl Forster. Followed by a work of Verdi, by the Philharmonic Orchestra, directed by Carlo Maria Giulini. On the second CD, you will find Francis Poulenc interpreted by Régine Crespin. This compilation ends with a little-known version, that of the Polish composer Karol Szymanowski. Polish National Radio Symphony Orchestra directed by Antoni Wit"
Double="true"
Duration="7246"
Type="Sacred music"
CD_ID="5" Performer="Various"
Price="8.5"
Title="4 Stabat mater"/>
```

```
Repeat
   MyEvent:=SAX Get XML node(DocRef)

   Case of
      :(MyEvent=XML Start Element)
         ARRAY TEXT(arrAttrNames;0)
         ARRAY TEXT(arrAttrValues;0)
         SAX GET XML ELEMENT(DocRef;vName;vPrefix;arrAttrNames;arrAttrValues)
         If(vName="CD")
            CREATE RECORD([CD])
            For($i;1;Size of array(arrAttrNames))
               $attrName:=arrAttrNames{$i}
               Case of
                  :($attrName="CD_ID")
                     XML DECODE(arrAttrValues{$i};[CD]CD_ID)
                  :($attrName="Title")
```

```
                    [CD]Work:=arrAttrValues{$i}
                :($attrName="Price")
                    XML DECODE(arrAttrValues{$i};[CD]Price)
                :($attrName="Date")
                    XML DECODE(arrAttrValues{$i};[CD]Date entered)
                :($attrName="Duration")
                    XML DECODE(arrAttrValues{$i};[CD]Total_duration)
                :($attrName="Double")
                    XML DECODE(arrAttrValues{$i};[CD]Double_CD)
            End case
        End for
    End if
    ...
End case
Until(MyEvent=XML End Document)
```

# XML GET ERROR

XML GET ERROR ( elementRef ; errorText {; row {; column}} )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇨ | XML element reference |
| errorText | Variable | ⇦ | Text of the error |
| row | Variable | ⇦ | Row number |
| column | Variable | ⇦ | Column number |

## Description

The **XML GET ERROR** command returns, in the *errorText* parameter, a description of the error encountered when processing the XML element designated by the *elementRef* parameter. The information returned is supplied by the Xerces.DLL library.

The optional *row* and *column* parameters indicate the location of the error: they retrieve, respectively, the row number and, in this row, the position of the first character of the expression at the origin of the error.

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. If an error occurs, it is set to 0.

# XML GET OPTIONS

XML GET OPTIONS ( elementRef | document ; selector ; value {; selector2 ; value2 ; ... ; selectorN ; valueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef \| document | Text, DocRef | ⇒ | XML root element reference or Reference of open document |
| selector | Longint | ⇒ | Option to get |
| value | Longint | ⇐ | Current value of option |

## Description

The XML GET OPTIONS command is used to get the current value of one or more XML parameters for the current session and the current user.

In selector, pass a constant of the "**XML**" theme indicating the option to get. The current value of the option is returned in the *value* parameter:

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| XML binary encoding | Longint | 5 | Specifies the way binary data will be converted.<br>**Possible values:**<br>• XML Base64 (default value): binary data are simply converted to Base64<br>• XML data URI scheme: binary data are converted to Base64 and the "data:;base64" header is added. This format mainly allows a browser to automatically decode a picture, and is also required for the insertion of pictures. For more information, see http://en.wikipedia.org/wiki/Data_URI_scheme. |
| XML date encoding | Longint | 2 | Specifies the way 4D dates will be converted. For example, !01/01/2003! in the Paris time zone.<br>**Possible values:**<br>• XML ISO (default value): use of the format xs:datetime without indication of time zone. Result: "2003-01-01". The time part, if it is present in the 4D value (via SQL) is lost.<br>• XML local: use of the format xs:date with indication of time zone. Result: "2003-01-01 +01:00". The time part, if it is present in the 4D value (via SQL) is lost.<br>• XML datetime local: use of the format xs:dateTime (ISO 8601). Indication of time zone. This format allows the time part to be kept, if it is present in the 4D value (via SQL). Result: "<Date>2003-01-01T00:00:00 +01:00</Date>".<br>• XML UTC: use of the format xs:date. Result: "2003-01-01Z". The time part, if it is present in the 4D value (via SQL) is lost.<br>• XML datetime UTC: use of the format xs:dateTime (ISO 8601). This format allows the time part to be kept, if it is present in the 4D value (via SQL). Result: "<Date>2003-01-01T00:00:00Z</Date>". |
| XML indentation | Longint | 4 | Specifies the indentation of the XML *document*.<br>**Possible values:**<br>• XML with indentation (default value): the document is indented.<br>• XML no indentation: the document is not indented; its contents are placed in a single line. |
| XML picture encoding | Longint | 6 | Specifies the way pictures must be converted (before encoding in Base64).<br>**Possible values:**<br>• XML convert to PNG (default value): pictures are converted to PNG before being encoded in Base64.<br>• XML native codec: pictures are converted in their first native storage CODEC before being encoded in Base64. You must use these options to encode SVG pictures (see example for the **XML SET OPTIONS** command). |
| XML string encoding | Longint | 1 | Specifies the way 4D strings are converted to element values. It does not concern the conversion to attributes for which XML imposes the use of escape characters.<br>**Possible values:**<br>• XML with escaping (default value): conversion of 4D strings to XML element values with replacement of characters. The Text type data are automatically parsed so that forbidden characters (<&>') are replaced by XML entities (&<>'").<br>• XML raw data: 4D strings are sent as raw data; 4D does not carry out encoding or parsing. 4D values are converted if possible to XML fragments and inserted as a child of the target element. If a value cannot be considered as an XML fragment, it is inserted as raw data into a new CDATA node. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| XML time encoding | Longint | 3 | Specifies the way 4D times are converted. For example, ?02/00/46? (Paris time). The encoding differs depending on whether you want to express a time or a duration.<br>**Possible values for times:**<br>• XML datetime UTC: time expressed in UTC (Universal Time Coordinated). Note that conversion to UTC is automatic. Result: "<Duration>0000-00-00T01:00:46Z</Duration>".<br>• XML datetime local: time expressed with the time difference of the machine of the 4D engine. Result: "<Duration>0000-00-00T02:00:46+01:00</Duration>".<br>• XML datetime local absolute (default value): time expressed without indication of time zone. No modification of the value. Result: "<Duration>0000-00-00T02:00:46</Duration>".<br>**Possible values for durations:**<br>• XML seconds: number of seconds since midnight; no modification of the value since it expresses a duration. Result: "<Duration>7246</Duration>".<br>• XML duration: duration expressed in compliance with XML Schema Part 2: Datatypes Second Edition. No modification of the value since it expresses a duration. Result: "<Duration>PT02H00M46S</Duration>". |

# XML SET OPTIONS

XML SET OPTIONS ( elementRef | document ; selector ; value {; selector2 ; value2 ; ... ; selectorN ; valueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef \| document | Text, DocRef | ⇒ | XML root element reference or Reference of open document |
| selector | Longint | ⇒ | Option to set |
| value | Longint | ⇒ | Value of option |

## Description

The **XML SET OPTIONS** command is used to modify the value of one or more XML options for the structure passed in the first parameter.

This command applies the XML structures of the "tree" type (DOM) or of the "document" type (SAX). In the first parameter, you can pass either a root element reference (*elementRef*), or the reference of an open SAX document (*document*).

The options set by this command are only used in the direction 4D to XML (it has no effect on the reading of XML values in 4D). The following commands use these options:

- **DOM SET XML ATTRIBUTE**
- **DOM SET XML ELEMENT VALUE**
- **SAX ADD XML ELEMENT VALUE**

Pass the option to be modified in *selector* and the new value of the option in *value*. You can pass as many *selector*/*value* pairs as you want.

You must use the constants described below, placed in the "**XML**" theme:

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| XML binary encoding | Longint | 5 | Specifies the way binary data will be converted.<br>**Possible values:**<br>• XML Base64 (default value): binary data are simply converted to Base64<br>• XML data URI scheme: binary data are converted to Base64 and the "data:;base64" header is added. This format mainly allows a browser to automatically decode a picture, and is also required for the insertion of pictures. For more information, see http://en.wikipedia.org/wiki/Data_URI_scheme. |
| XML date encoding | Longint | 2 | Specifies the way 4D dates will be converted. For example, !01/01/2003! in the Paris time zone.<br>**Possible values:**<br>• XML ISO (default value): use of the format xs:datetime without indication of time zone. Result: "2003-01-01". The time part, if it is present in the 4D value (via SQL) is lost.<br>• XML local: use of the format xs:date with indication of time zone. Result: "2003-01-01 +01:00". The time part, if it is present in the 4D value (via SQL) is lost.<br>• XML datetime local: use of the format xs:dateTime (ISO 8601). Indication of time zone. This format allows the time part to be kept, if it is present in the 4D value (via SQL). Result: "<Date>2003-01-01T00:00:00 +01:00</Date>".<br>• XML UTC: use of the format xs:date. Result: "2003-01-01Z". The time part, if it is present in the 4D value (via SQL) is lost.<br>• XML datetime UTC: use of the format xs:dateTime (ISO 8601). This format allows the time part to be kept, if it is present in the 4D value (via SQL). Result: "<Date>2003-01-01T00:00:00Z</Date>". |
| XML indentation | Longint | 4 | Specifies the indentation of the XML *document*.<br>**Possible values:**<br>• XML with indentation (default value): the document is indented.<br>• XML no indentation: the document is not indented; its contents are placed in a single line. |
| XML picture encoding | Longint | 6 | Specifies the way pictures must be converted (before encoding in Base64).<br>**Possible values:**<br>• XML convert to PNG (default value): pictures are converted to PNG before being encoded in Base64.<br>• XML native codec: pictures are converted in their first native storage CODEC before being encoded in Base64. You must use these options to encode SVG pictures (see example for the **XML SET OPTIONS** command). |
| XML string encoding | Longint | 1 | Specifies the way 4D strings are converted to element values. It does not concern the conversion to attributes for which XML imposes the use of escape characters.<br>**Possible values:**<br>• XML with escaping (default value): conversion of 4D strings to XML element values with replacement of characters. The Text type data are automatically parsed so that forbidden characters (<&>') are replaced by XML entities (&<>'").<br>• XML raw data: 4D strings are sent as raw data; 4D does not carry out encoding or parsing. 4D values are converted if possible to XML fragments and inserted as a child of the target element. If a value cannot be considered as an XML fragment, it is inserted as raw data into a new CDATA node. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| XML time encoding | Longint | 3 | Specifies the way 4D times are converted. For example, ?02/00/46? (Paris time). The encoding differs depending on whether you want to express a time or a duration.<br>**Possible values for times:**<br>• XML datetime UTC: time expressed in UTC (Universal Time Coordinated). Note that conversion to UTC is automatic. Result: "<Duration>0000-00-00T01:00:46Z</Duration>".<br>• XML datetime local: time expressed with the time difference of the machine of the 4D engine. Result: "<Duration>0000-00-00T02:00:46+01:00</Duration>".<br>• XML datetime local absolute (default value): time expressed without indication of time zone. No modification of the value. Result: "<Duration>0000-00-00T02:00:46</Duration>".<br>**Possible values for durations:**<br>• XML seconds: number of seconds since midnight; no modification of the value since it expresses a duration. Result: "<Duration>7246</Duration>".<br>• XML duration: duration expressed in compliance with XML Schema Part 2: Datatypes Second Edition. No modification of the value since it expresses a duration. Result: "<Duration>PT02H00M46S</Duration>". |

**Notes:**

- The XML local and XML datetime local values do not provide dates expressed in UTC (Universal Time Coordinated); they are converted without modification but indicating the time difference. These formats are useful in the case of successive and reciprocal conversions (round tripping).
- The XML UTC and XML datetime UTC values are equivalent to the previous from the formatting viewpoint, but are expressed in UTC. These formats should be given priority to ensure interoperability. The values are not modified.

## Example

Insertion of an SVG picture:

```
XML SET OPTIONS($pictElemRef;XML binary encoding;XML data URI scheme)
XML SET OPTIONS($pictElemRef;XML picture encoding;XML native codec)
DOM SET XML ATTRIBUTE($pictElemRef;"xlink:href";PictVar)
```

# _o_XSLT APPLY TRANSFORMATION

_o_XSLT APPLY TRANSFORMATION ( xmlSource ; xslSheet ; result {; compileSheet} )

| Parameter | Type | | Description |
|---|---|---|---|
| xmlSource | String, BLOB | ⇒ | Name or access path of XML source document, or BLOB containing the XML source |
| xslSheet | String, BLOB | ⇒ | Name or access path of document containing XSL stylesheet, or BLOB containing the XSL stylesheet |
| result | String, BLOB | ⇒ | Name or access path of the document receiving the result of the XSLT transformation, or BLOB receiving the result of the XSLT transformation |
| compileSheet | Boolean | ⇒ | True = Optimize XSLT transformation False or omitted = No optimization, remove the compiled XSL file (if any) |

## Compatibility note

*Starting with 4D v14 R4, XSL transformation commands are obsolete. For compatibility, they are still supported in 4D but we strongly recommend that you discontinue using them. In future versions of 4D, it will no longer be possible to use XSLT technology. For more information, please refer to the* **Overview of XML Utilities Commands***.*

# _o_XSLT GET ERROR

_o_XSLT GET ERROR ( errorText {; row {; column}} )

| Parameter | Type | | Description |
|---|---|---|---|
| errorText | Variable | ⇦ | Text of the error |
| row | Variable | ⇦ | Row number |
| column | Variable | ⇦ | Column number |

## Compatibility note

*Starting with 4D v14 R4, XSL transformation commands are obsolete. For compatibility, they are still supported in 4D but we strongly recommend that you discontinue using them. In future versions of 4D, it will no longer be possible to use XSLT technology. For more information, please refer to the* **Overview of XML Utilities Commands**.

# _o_XSLT SET PARAMETER

_o_XSLT SET PARAMETER ( paramName ; paramValue )

| Parameter | Type | | Description |
|---|---|---|---|
| paramName | String | ➡ | Name of the parameter to look for in the XSL sheet |
| paramValue | String | ➡ | Value of the parameter to use in the transformed document |

## Compatibility note

*Starting with 4D v14 R4, XSL transformation commands are obsolete. For compatibility, they are still supported in 4D but we strongly recommend that you discontinue using them. In future versions of 4D, it will no longer be possible to use XSLT technology. For more information, please refer to the* **Overview of XML Utilities Commands***.*

# XML DOM

- Overview of XML DOM Commands
- DOM Append XML child node
- DOM Append XML element
- DOM CLOSE XML
- DOM Count XML attributes
- DOM Count XML elements
- DOM Create XML element
- DOM Create XML element arrays
- DOM Create XML Ref
- DOM EXPORT TO FILE
- DOM EXPORT TO VAR
- DOM Find XML element
- DOM Find XML element by ID
- DOM Get first child XML element
- DOM Get last child XML element
- DOM Get next sibling XML element
- DOM Get parent XML element
- DOM Get previous sibling XML element
- DOM Get Root XML element
- DOM GET XML ATTRIBUTE BY INDEX
- DOM GET XML ATTRIBUTE BY NAME
- DOM GET XML CHILD NODES
- DOM Get XML document ref
- DOM Get XML element
- DOM GET XML ELEMENT NAME
- DOM GET XML ELEMENT VALUE
- DOM Get XML information
- DOM Insert XML element
- DOM Parse XML source
- DOM Parse XML variable
- DOM REMOVE XML ATTRIBUTE
- DOM REMOVE XML ELEMENT
- DOM SET XML ATTRIBUTE
- DOM SET XML DECLARATION
- DOM SET XML ELEMENT NAME
- DOM SET XML ELEMENT VALUE

# Overview of XML DOM Commands

4D includes a set of commands used for parsing objects containing XML (eXtensible Markup Language) data.

**Note concerning preemptive mode:** XML references created by a preemptive process can only be used in that specific process. Conversely, XML references created by a cooperative process can be used by any other cooperative process, but cannot be used by any preemptive process.

## About the XML language

The XML language is a data exchange standard. It is based on the use of tags and enables precise description of the data exchanged as well as their structure. XML files are Text format files; their content is parsed by the applications importing the data. Many applications now support this format.

For more information about XML, refer, for instance, to the http://xml.org and http://www.w3.org sites.

For XML support, 4D uses a library named Xerces.dll developed by the Apache Foundation company. 4D supports XML version 1.0.

**Note**: 4D allows direct importing and exporting of data in XML format using the import/export editor.

## DOM and SAX

The commands of this theme are prefixed DOM. In fact, 4D offers two separate sets of XML commands, prefixed DOM and SAX: DOM (Document Object Model) and SAX (Simple API XML) are two different parsing modes for XML documents.

- The DOM mode parses an XML source and builds its structure (its “tree”) in memory. Because of this, access to each element of the source is extremely fast. However, since the entire tree structure is stored in memory, the processing of large XML documents may lead to the memory capacity being exceeded and thus provoke errors.
- The SAX mode does not build a tree structure in memory. In this mode, “events” (such as the start and end of an element) are generated when parsing the source. This mode lets you parse XML documents of any size, regardless of the amount of memory available. The SAX commands are grouped together in the "**XML SAX**" theme. For more information, please refer to the **Overview of XML SAX Commands** section.

For more information on XML standards, consult the following sites: http://www.saxproject.org/?selected=event and http://www.w3schools.com/xml/.

## Creating, opening and closing XML documents via DOM

Objects created, modified or parsed by the 4D DOM commands can be text, URLs, documents or BLOBs. The DOM commands used for opening XML objects in 4D are **DOM Parse XML source** and **DOM Parse XML variable**.

Many commands then let you read, parse and write the elements and attributes. Errors are recovered using the **DOM Parse XML variable** command (common to both XML standards).

The **XML GET ERROR** command lets you close the source in the end.

**Note about use of XML BLOB parameters:** XML structures are based on Text type data so it is recommended to use Text type fields or variables to work with them. For historical reasons, the XML commands of 4D (for example **DOM Parse XML variable**) accept BLOB type parameters. In previous versions of 4D, the size of Text type variables was limited to 32 KB. Starting with version 11 of 4D, Text fields and variables can contain up to 2 GB of data. Since the previous limitation was removed, it is now highly inadvisable to store text in BLOBs. The use of BLOBs is reserved for processing binary data. In conformity with XML specifications, beginning with 4D v12, binary data are automatically encoded in Base64, even when the BLOB contains text.

## Use of XPath notation (DOM)

Three XML DOM commands (**DOM Create XML element**, **DOM Find XML element** and **DOM SET XML ELEMENT VALUE**) accept XPath notation for accessing XML elements.
XPath notation comes from the XPath language, designed to navigate within XML structures. It allows the setting of elements directly within an XML structure via a "pathname" type syntax, without necessarily having to indicate the complete pathname in order to reach it. For example, given the following structure:

```
<RootElement>      <Elem1>          <Elem2>            <Elem3 Font=Verdana Size=10> </Elem3>          </Elem2>
   </Elem1>    </RootElement>
```

XPath notation allows you to access element 3 using the */RootElement/Elem1/Elem2/Elem3* syntax.

4D also accepts **indexed** XPath elements using the *Element[ElementNum]* syntax. For example, given the following structure:

```
<RootElement>      <Elem1>          <Elem2>aaa</Elem2>          <Elem2>bbb</Elem2>          <Elem2>ccc</Elem2>
   </Elem1>    </RootElement>
```

XPath notation allows you to access the “ccc” value using the */RootElement/Elem1/Elem2[3]* syntax.

For an illustration of XPath notation, please refer to the examples in the **DOM Create XML element** and **DOM Find XML element** commands.

## Character Sets

The following character sets are supported by the XML DOM and XML SAX commands of 4D:

- ASCII
- UTF-8
- UTF-16 (Big/Small Endian)
- UCS4 (Big/Small Endian)
- EBCDIC code pages IBM037, IBM1047 and IBM1140 encodings,
- ISO-8859-1 (or Latin1)
- Windows-1252.

## Terminology

The XML language uses a number of specific terms and acronyms. This non-exhaustive list details the main XML concepts used by the commands and functions of 4D.

**Attribute**: an XML sub-tag associated with an element. An attribute always contains a name and a value (see diagram below).

**Child**: In an XML structure, an element in a level directly below another.

**DTD**: Document Type Declaration The DTD records the set of specific rules and properties that the XML must follow. These rules define, more particularly, the name and content of each tag as well as its context. This formalization of the elements can be used to check whether an XML document is in compliance (in which case, it is declared “valid”).
The DTD may be included in the XML document (internal DTD) or in a separate document (external DTD). Note that the DTD is not mandatory.

**Element**: an XML tag. An element always contains a name and a value. Optionally, an element may contain attributes (see diagram).


**ElementRef**: XML reference used by the 4D XML commands to specify an XML structure. This reference is made up of 8 coded characters in hexadecimal form, which means that its length is either 16 or 32 characters depending on whether you use a 32- or 64-bit system. It is recommended to declare XML references using the **C_TEXT** directive.

**Parent**: In an XML structure, an element in a level directly above another.

**Parsing**, **parser**: The act of analyzing the contents of a structured object in order to extract useful information. The commands of the “XML” theme are used to parse the contents of any XML objects.

**Root**: An element located at the first level of an XML structure.

**Sibling**: In an XML structure, an element at the same level as another.

**Structure XML**: structured XML object. This object can be a document, a variable, or an element.

**Validation**: An XML document is “validated” by the parser when it is “well-formed” and in compliance with the DTD specifications. See also Well-formed.

**Well-formed**: An XML document is declared “well-formed” by the parser when it complies with the generic XML specifications. See also Validation.

**XML**: eXtensible Markup Language. A computerized data exchange standard enabling the transfer of data as well as their structure. The XML language is based on the use of tags and a specific syntax, in keeping with the HTML language. However, unlike the latter, the XML language allows the definition of customized tags.

**XSL**: eXtensible Stylesheet Language. A language permitting the definition of style sheets used to process and display the contents of an XSL document.

## Error Handling

Many functions in this theme return an XML element reference. If an error occurs during function execution (for example, if the root element reference is not valid), the OK variable is set to 0 and an error is generated.

In addition, the reference returned in this case is a sequence of zero "0" characters (16 characters in 32 bits, and 32 characters in 64 bits).

# DOM Append XML child node

DOM Append XML child node ( elementRef ; childType ; childValue ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | Text | ⇒ | XML element reference |
| childType | Longint | ⇒ | Type of child to append |
| childValue | Text, BLOB | ⇒ | Text or variable (Text or BLOB) whose value must be inserted as child node |
| Function result | Text | ⇐ | Reference of child XML element |

## Description

The **DOM Append XML child node** command is used to append the *childValue* value to the XML node designated by *elementRef*.

The type of node created is specified by the *childType* parameter. In this parameter you can pass one of the following constants, located in the "**XML**" theme:

| Constant | Type | Value |
|---|---|---|
| XML CDATA | Longint | 7 |
| XML comment | Longint | 2 |
| XML DATA | Longint | 6 |
| XML DOCTYPE | Longint | 10 |
| XML ELEMENT | Longint | 11 |
| XML processing instruction | Longint | 3 |

In *childValue*, pass the data to be inserted. You can pass a string or a 4D variable (string or BLOB). The contents of this parameter will always be converted into text.

**Note:** If the *elementRef* parameter designates the Document node (top level node), the command inserts a "Doctype" node before any other node. The same goes for processing instructions and comments, which are always inserted before the root node (but after the Doctype node).

## Example 1

Adding a text type node:

```
Reference:=DOM Create XML element(elementRef;"myElement")
DOM SET XML ELEMENT VALUE(Reference;"Hello")
temp:=DOM Create XML element(Reference;"br")
temp:=DOM Append XML child node(Reference;XML DATA;"New")
temp:=DOM Create XML element(Reference;"br")
temp:=DOM Append XML child node(Reference;XML DATA;"York")
```

Result:

```
<myElement>Hello<br/>New<br/>York</myElement>
```

## Example 2

Adding a processing instruction type node:

```
$Txt_instruction:="xml-stylesheet type = ¥"text/xsl¥" href=¥"style.xsl¥""
Reference:=DOM Append XML child node(elementRef;XML Processing Instruction;$Txt_instruction)
```

Result (inserted before first element):

```
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
```

## Example 3

Adding a comment type node:

```
Reference:=DOM Append XML child node(elementRef;XML_Comment;"Hello world")
```

Result:

```
<!--Hello world-->
```

## Example 4

Adding a CDATA type node:

```
Reference:=DOM Append XML child node(elementRef;XML_CDATA;"12 < 18")
```

Result:

```
<element><![CDATA[12 < 18]]></element>
```

## Example 5

Adding or replacing a Doctype declaration type node:

```
Reference:=DOM Append XML child node(elementRef;XML_DOCTYPE;"Books SYSTEM ¥"Book.DTD¥"")
```

Result (inserted before first element):

```
<!DOCTYPE Books SYSTEM "Book.DTD">
```

## Example 6

Adding or replacing an Element type node.

- if the *childValue* parameter is an XML fragment, it is inserted as child nodes:

  ```
  Reference:=DOM Append XML child node(elementRef;XML_ELEMENT;"<child>simon</child><child>eva</child>")
  ```

  Result:

  ```
  <parent>    <child>simon</child>    <child>eva</child> </parent>
  ```

- otherwise, a new blank child element is appended:

  ```
  Reference:=DOM Append XML child node(elementRef;XML_ELEMENT;"tbreak")
  ```

  Result:

  ```
  <parent>    <tbreak/> </parent>
  ```

If the contents of *childValue* are not valid, an error is returned.

# DOM Append XML element

DOM Append XML element ( targetElementRef ; sourceElementRef ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| targetElementRef | Text | ⇒ | Reference of XML parent element |
| sourceElementRef | Text | ⇒ | Reference of XML element to append |
| Function result | Text | ⇐ | Reference of new XML element |

## Description

The **DOM Append XML element** command is used to add a new XML element to the children of the XML element whose reference is passed in the *targetElementRef* parameter.

In the *sourceElementRef* parameter, pass the element to be added. This element must be passed as the reference of an existing XML element in a DOM tree. It is added after the last existing child element of *targetElementRef*.

## Example

See the example of the **DOM Insert XML element** command.

# DOM CLOSE XML

DOM CLOSE XML ( elementRef )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇨ | XML root element reference |

## Description

The **DOM CLOSE XML** command frees up the memory occupied by the XML object designated by *elementRef*.
If *elementRef* is not an XML root object, an error is generated.

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. If an error occurs, it is set to 0.

# DOM Count XML attributes

DOM Count XML attributes ( elementRef ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ➡ | XML element reference |
| Function result | Longint | ⮌ | Number of attributes |

## Description

The **DOM Count XML attributes** command returns the number of XML attributes present in the XML element designated by *elementRef*. For more information about XML attributes, refer to the **Overview of XML DOM Commands** section.

## Example

Before retrieving the values of elements in an array, you want to know the number of attributes in the following XML element:

```
<?xml version="1.0" ?>
- <LINES>
  <LINE N="1" Font="Verdana" size="10">this is a line</LINE>
  <LINE N="2" Font="charcoal" size="12">and another line</LINE>
  <LINE N="3" Font="Verdana" size="18">and we stop here</LINE>
</LINES>
```

```
C_BLOB(myBlobVar)
C_TEXT($xml_Parent_Ref;$xml_Child_Ref)
C_TEXT(myResult)
C_LONGINT($numAttributes)

$xml_Parent_Ref:=DOM Parse XML variable(myBlobVar)
$xml_Child_Ref:=DOM Get first child XML element($xml_Parent_Ref)

$numAttributes:=DOM Count XML attributes($xml_Child_Ref)
ARRAY TEXT(tAttrib;$numAttributes)
ARRAY TEXT(tValAttrib;$numAttributes)
For($i;1;$numAttributes)
   DOM GET XML ATTRIBUTE BY INDEX($xml_Child_Ref;$i;tAttrib{$i};tValAttrib{$i})
End for
```

In the above example, $numAttributes equals 3, tAttrib{1} contains “Font”, tAttrib{2} contains “N”, tAttrib{3} contains “size” and tValAttrib contains “Verdana”, “1” and “10”.

**Note:** The index number does not correspond to the location of the attribute in the XML file displayed in text form. In XML, the index of an attribute indicates its position among the attributes arranged in alphabetical order (according to their name).

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. If an error occurs, it is set to 0.

# DOM Count XML elements

DOM Count XML elements ( elementRef ; elementName ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| elementName | String | ⇒ | Name of XML elements to count |
| Function result | Longint | ⇐ | Number of elements |

## Description

The **DOM Count XML elements** command returns the number of “child” elements dependent on the *elementRef* parent element and named *elementName*.

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. If an error occurs, it is set to 0.

# DOM Create XML element

DOM Create XML element ( elementRef ; xPath {; attribName ; attrValue} {; attribName2 ; attrValue2 ; ... ; attribNameN ; attrValueN} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | Root XML element reference |
| xPath | Text | ⇒ | XPath path of the XML element to create |
| attribName | String | ⇒ | Attribute to set |
| attrValue | String, Boolean, Longint, Real, Time, Date | ⇒ | New attribute value |
| Function result | String | ⇐ | Reference of the created XML element |

## Description

The **DOM Create XML element** command creates a new element in the XML element *elementRef* in the path set by the *xPath* parameter and adds attributes to it if necessary.

Pass the root element reference in *elementRef* (created, for example, using the **DOM Create XML Ref** command).

In *xPath*, pass the access path in XPath notation of the element to create (see the “Use of XPath notation” paragraph in the **Overview of XML DOM Commands** section). If any path elements do not exist, they are created.

It is possible to pass a simple item name directly in the *xPath* parameter in order to create a sub-item from the current item (see example 3).

**Note:** If you have defined one or more namespaces for the tree set using *elementRef* (see the **DOM Create XML Ref** command), you must precede the *xPath* parameter with the namespace to be used (for example, “MyNameSpace:MyElement”).

You can pass attribute/attribute value pairs (in the form of variables, fields or literal values) in the optional *attrName* and *attrValue* parameters. You can pass as many pairs as you want.

The *attrValue* parameter can be of the text type or another type (Boolean, integer, real, date or time). If you pass a value other than text, 4D handles its conversion to text, according to the following principles:

| Type | Example of converted value |
|---|---|
| Boolean | "true" or "false" |
| Integer | "123456" |
| Real | "12.34" (the decimal separator is always ".") |
| Date | "2006-12-04T00:00:00Z" (RFC 3339 standard) |
| Time | "5233" (number of seconds) |

The command returns the XML reference of the element created as a result.

## Example 1

We want to create the following element:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?> <RootElement>    <Elem1>        <Elem2>            <Elem3> </Elem3>
        <Elem3> </Elem3>        </Elem2>    </Elem1> </RootElement>
```

To do so, simply write:

```
C_TEXT(vRootRef;vElemRef)
vRootRef:=DOM Create XML Ref("RootElement")
vxPath:="/RootElement/Elem1/Elem2/Elem3"
vElemRef:=DOM Create XML element(vRootRef;vxPath)
vxPath:="/RootElement/Elem1/Elem2/Elem3[2]"
vElemRef:=DOM Create XML element(vRootRef;vxPath)
```

## Example 2

We want to create the following element (containing attributes):

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?> <RootElement>    <Elem1>       <Elem2>          <Elem3 Font=Verdana Size=10> </Elem3>       </Elem2>    </Elem1> </RootElement>
```

To do so, simply write:

```
C_TEXT(vRootRef;vElemRef)
C_TEXT($aAttrName1;$aAttrName2;$aAttrVal1;$aAttrVal2)
$aAttrName1:="Font"
$aAttrName2:="Size"
$aAttrVal1:="Verdana"
$aAttrVal2:="10"

vRootRef:=DOM Create XML Ref("RootElement")
vxPath:="/RootElement/Elem1/Elem2/Elem3"
vElemRef:=DOM Create XML element(vRootRef;vxPath;$aAttrName1;$aAttrVal1;$aAttrName2;$aAttrVal2)
```

## Example 3

We want to create and export the following structure:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?> <Root>    <Elem1>Hello</Elem1> </Root>
```

We want to use the syntax based on a simple item name. To do this, simply write:

```
C_TEXT($root)
C_TEXT($ref1)

$root:=DOM Create XML Ref("Root")
$ref1:=DOM Create XML element($root;"Elem1")
DOM SET XML ELEMENT VALUE($ref1;"Hello")
DOM EXPORT TO FILE($root;"mydoc.xml")
DOM CLOSE XML($root)
```

## System variables and sets

If the command was executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

## Error management

An error is generated when:

- The root element reference is invalid.
- The name of the element to create is invalid (for example, if it starts with a number).

# DOM Create XML element arrays

DOM Create XML element arrays ( elementRef ; xPath {; attribNamesArray ; attribValuesArray} {; attribNamesArray2 ; attribValuesArray2 ; ... ; attribNamesArrayN ; attribValuesArrayN} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | Text | ⇒ | XML root element reference |
| xPath | Text | ⇒ | XPath path of the XML element to create |
| attribNamesArray | String array | ⇒ | Array of attribute names |
| attribValuesArray | String array | ⇒ | Array of attribute values |
| Function result | Text | ⇐ | Reference of created XML element |

## Description

The **DOM Create XML element arrays** command is used to add a new element in the *elementRef* XML element, as well as, optionally, attributes and their values in the form of arrays.

Apart from supporting arrays (see below), this command is identical to **DOM Create XML element**. Please refer to the description of this command for the details of its functioning.

Optionally, this command can be used to pass several pairs of attributes and attribute values as arrays in the *attribNamesArray* and *attribValuesArray* parameters. In *attribValuesArray*, you can pass arrays of the text, date, number, and picture type. 4D automatically carries out the necessary conversions; you can modify these conversions using the **XML SET OPTIONS** command.

The arrays must have been created beforehand and function by pairs. You can pass as many pairs of arrays and as many elements in each pair as you want.

## Example

We want to create the following element:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?> <RootElement>    <Elem1>       <Elem2>          <Elem3
Font="Verdana" Size="10" Style="Bold"></Elem3>       </Elem2>    </Elem1> </RootElement>
```

For this, you can simply write:

```
ARRAY TEXT(arrAttNames;3)
ARRAY TEXT(arrAttValues;3)
arrAttNames{1}:="Font"
arrAttValues{1}:="Verdana"
arrAttNames{2}:="Size"
arrAttValues{2}:="10"
arrAttNames{3}:="Style"
arrAttValues{3}:="Bold"
vRootRef:=DOM Create XML Ref("RootElement")
vxPath:="/RootElement/Elem1/Elem2/Elem3"
vElementRef:=DOM Create XML element arrays(vRootRef;vxPath;arrAttNames;arrAttValues)
```

# DOM Create XML Ref

DOM Create XML Ref ( root {; nameSpace} {; nameSpaceName ; nameSpaceValue} {; nameSpaceName2 ; nameSpaceValue2 ; ... ; nameSpaceNameN ; nameSpaceValueN} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| root | String | ➡ | Name of root element |
| nameSpace | String | ➡ | Value of namespace |
| nameSpaceName | String | ➡ | Namespace name |
| nameSpaceValue | String | ➡ | Namespace value |
| Function result | String | ⬅ | Root XML element reference |

## Description

The **DOM Create XML Ref** command creates an empty XML tree in memory and returns its reference.

Pass the name of the XML tree root element in the *root* parameter.

Pass the declaration of the namespace value of the tree in the optional *nameSpace* parameter (for example, “http://www.4d.com”).
Note that you can prefix the *root* parameter with the namespace name followed by a colon *:* (for example “MyNameSpace:MyRoot”). In this case, the *nameSpace* parameter specifying the namespace value is mandatory.

**Note:** The namespace is a string that allows you to make sure the XML variable names are unique. In general, a URL like http://www.mysite.com/myurl is used. The URL does not necessarily have to be valid, but it does have to be unique.

You can declare one or more additional namespaces in the generated XML tree using *nameSpaceName/nameSpaceValue* pairs. You can pass as many namespace name/value pairs as you want.

**Important:** Remember to call the **DOM CLOSE XML** command in order to free up the memory when you have finished using the XML tree.

## Example 1

Creating a single XML tree:

```
C_TEXT(vElemRef)
vElemRef:=DOM Create XML Ref("MyRoot")
```

This code produces the following result:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?> <MyRoot/>
```

## Example 2

Creating an XML tree with a single namespace:

```
C_TEXT(vElemRef)
$Root:="MyNameSpace:MyRoot"
$Namespace:="http://www.4D.com/tech/namespace"
vElemRef:=DOM Create XML Ref($Root;$Namespace)
```

This code produces the following result:

```
<MyNameSpace:MyRoot xmlns:MyNameSpace="http://www.4D.com/tech/namespace"/>
```

## Example 3

Creating an XML tree with several namespaces:

```
C_TEXT(vElemRef)
C_TEXT($aNSName1;$aNSName2;$aNSValue1;$aNSValue2)
$Root:="MyNameSpace:MyRoot"
$Namespace:="http://www.4D.com/tech/namespace"
$aNSName1:="NSName1"
$aNSName2:="NSName2"
$aNSValue1:="http://www.4D.com/Prod/namespace"
$aNSValue2:="http://www.4D.com/Mkt/namespace"
vElemRef:=DOM Create XML Ref($Root;$Namespace;$aNSName1;$aNSValue1;$aNSName2;$aNSValue2)
```

This code produces the following result:

```
<MyNameSpace:MyRoot xmlns:MyNameSpace="http://www.4D.com/tech/nameSpace" NSName1="http://www.4D.com/Prod/namespace"
NSName2="http://www.4D.com/Mkt/namespace"/>
```

## System variables and sets

If the command was executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# DOM EXPORT TO FILE

DOM EXPORT TO FILE ( elementRef ; filePath )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | Root XML element reference |
| filePath | Text | ⇒ | Full access path of the file |

## Description

The **DOM EXPORT TO FILE** command stores an XML tree in a file on disk.

Pass the root element reference to export in the *elementRef* parameter.

Pass the full access path to use or create of the export file in *filePath*. If the file does not already exist, it is created.

If you only pass a file name (without an access path), a search for the file will take place or it will be created next to the structure file.

If you pass an empty string (""), a standard open file and new file dialog box appears.

### Notes about processing end-of-line characters

In XML, line breaks are not significant regardless of whether they are within or between XML elements. Internally, XML uses standard LF characters as line separators.

During import and export operations, line break characters can be converted. During an import, the XML parser replaces CRLF characters (standard line breaks under Windows) with LF characters. During export, LF characters are replaced by CR characters.

If you want to keep carriage returns, you must include them in an XML CDATA element so that they will not be processed by the XML parser. Instead of CRLF characters, you can also use "<br/>" characters, which are explicit carriage returns that will not be processed by the parser.

## Example

This example stores the tree *vElemRef* in the file MyDoc.xml:

```
DOM EXPORT TO FILE(vElemRef;"C:¥¥folder¥MyDoc.xml")
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

## Error management

An error is generated when:

- The element reference is invalid,
- The specified access path is invalid,
- The storage volume returns an error (insufficient disk space, etc.).

# DOM EXPORT TO VAR

DOM EXPORT TO VAR ( elementRef ; vXmlVar )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | Root XML element reference |
| vXmlVar | Text, BLOB | ⇒ | Variable to receive XML tree |

## Description

The **DOM EXPORT TO VAR** command saves an XML tree in a text or BLOB variable.

Pass the root element reference to export in *elementRef*.

Pass the name of the variable that must contain the XML tree in *vXmlVar*. This variable must either be a Text or BLOB type. You can select the type depending on what you plan on doing next or the size that the tree can reach (remember that when not in Unicode mode, Text type variables are limited to 32 K of text, whereas in Unicode mode, this limit is 2 GB).

Keep in mind that if you use a Text variable to store *elementRef* when not in Unicode mode, , it will be encoded using the “current” Mac character set (i.e. Mac Roman on most Western systems). This means that the text returned will lose its original encoding (encoding="xxx"). In this case, the *vVarXml* variable allows you to view or store the code but NOT to generate a valid XML document (using the **SEND PACKET** command for example).

In Unicode mode, the original encoding is kept in the variable.

### Notes about processing end-of-line characters

In XML, line breaks are not significant regardless of whether they are within or between XML elements. Internally, XML uses standard LF characters as line separators.

During import and export operations, line break characters can be converted. During an import, the XML parser replaces CRLF characters (standard line breaks under Windows) with LF characters. During export, LF characters are replaced by CR characters.

If you want to keep carriage returns, you must include them in an XML CDATA element so that they will not be processed by the XML parser. Instead of CRLF characters, you can also use "<br/>" characters, which are explicit carriage returns that will not be processed by the parser.

## Example

This example stores the tree vElemRef in a text variable:

```
C_TEXT(vtMyText)
DOM EXPORT TO VAR(vElemRef;vtMyText)
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated (for example, if the element reference is invalid).

# DOM Find XML element

DOM Find XML element ( elementRef ; xPath {; arrElementRefs} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| xPath | Text | ⇒ | XPath path of the element to look for |
| arrElementRefs | String array | ⇐ | List of element references found (if applicable) |
| Function result | String | ⇒ | Reference of the element found (if applicable) |

## Description

The **DOM Find XML element** command looks for specific XML elements in an XML structure. The search starts at the element designated by the *elementRef* parameter.

The XML node to seek is set expressed in XPath notation using the *xPath* parameter (see the "Use of XPath notation" parameter in the **Overview of XML DOM Commands** section). Indexed elements can be used.

**Note:** In conformity with the XML standard, searches will be case sensitive.

The command returns the XML reference of the element found.

When the *arrElementRefs* string array is passed, the command fills it with the list of XML references found. In this case, the command returns the first element of the *arrElementRefs* array as the result. This parameter is useful when several elements with the same name exist at the location specified by the *xPath* parameter.

## Example 1

This example lets you quickly look for an XML element and display its value:

```
vFound:=DOM Find XML element(vElemRef;"Items/Book[15]/Title")
DOM GET XML ELEMENT VALUE(vFound;value)
ALERT("The value of the element is: ¥""+value+"¥"")
```

The same search can also be done as follows:

```
vFound:=DOM Find XML element(vElemRef;"Items/Book[15]")
vFound:=DOM Find XML element(vFound;"Book/Title")
DOM GET XML ELEMENT VALUE(vFound;value)
ALERT("The value of the element is: ¥""+value+"¥"")
```

**Note:** As you can see in the above example, the XPath path must always begin with the name of the current element. This detail is important when you are handling relative XPath paths.

## Example 2

Given the following XML structure:

```
<Root>  <Elem1>  <Elem2>aaa</Elem2>  <Elem2>bbb</Elem2>  <Elem2>ccc</Elem2>  </Elem1> </Root>
```

The following code can be used to retrieve the reference of each Elem2 element in the arrAfound array:

```
ARRAY TEXT(arrAfound;0)
vFound:=DOM Find XML element(vElemRef;"/Root/Elem1/Elem2";arrAfound)
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

## Error management

An error is generated when:

- The element reference is invalid
- The specified xPath path is invalid.

# DOM Find XML element by ID

DOM Find XML element by ID ( elementRef ; id ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| id | String | ⇒ | Value of ID attribute of element to look for |
| Function result | String | ⇐ | Reference of the element found (if applicable) |

## Description

The **DOM Find XML element by ID** command searches within an XML document for the element whose id attribute equals the value passed in the *id* parameter.

In *elementRef*, pass the reference of an element in the XML document where you want to perform the search. You can pass the reference of the root element or any other element; the search does not take the position of *elementRef* into account and always searches the whole document.

The command returns the XML reference of the element found as a result.

**Warning:** In XML, the id attribute associates a unique ID to each document element. The value of the id attribute must be a valid XML name and it must be unique among all the elements in the XML document (validity constraint). In order for the **DOM Find XML element by ID** command to work properly, this constraint must be complied with; otherwise, the result is random (the command returns the reference to the first element found in the document).

# DOM Get first child XML element

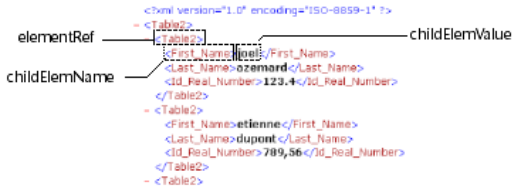
DOM Get first child XML element ( elementRef {; childElemName {; childElemValue}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| childElemName | String | ⇐ | Name of child XML element |
| childElemValue | String | ⇐ | Value of child XML element |
| Function result | String | ⇒ | Child XML element reference (16 characters) |

## Description

The **DOM Get first child XML element** command returns a reference to the first “child” of the XML element passed in *elementRef*. This reference can be used with other XML parsing commands.

The *childElemName* and *childElemValue* parameters, if they are passed, receive respectively the name and the value of the child element.




## Example 1

Retrieval of the reference of the first XML element of the parent root. The XML structure (C:¥¥import.xml) is first loaded into a BLOB:

```
C_BLOB(myBlobVar)
C_TEXT($xml_Parent_Ref;$xml_Child_Ref)

DOCUMENT TO BLOB("c:¥¥import.xml";myBlobVar)
$xml_Parent_Ref:=DOM Parse XML variable(myBlobVar)
$xml_Child_Ref:=DOM Get first child XML element($xml_Parent_Ref)
```

## Example 2

Retrieval of the reference, name and value of the first XML element of the parent root. The XML structure (C:¥¥import.xml) is first loaded into a BLOB:

```
C_BLOB(myBlobVar)
C_TEXT($xml_Parent_Ref;$xml_Child_Ref)
C_TEXT($childName;$childValue)

DOCUMENT TO BLOB("c:¥¥import.xml";myBlobVar)
$xml_Parent_Ref:=DOM Parse XML variable(myBlobVar)
$xml_Child_Ref:=DOM Get first child XML element($xml_Parent_Ref;$childName;$childValue)
```

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. Otherwise, it is set to 0.

# DOM Get last child XML element

DOM Get last child XML element ( elementRef {; childElemName {; childElemValue}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| childElemName | String | ⇐ | Name of child element |
| childElemValue | String | ⇐ | Value of child element |
| Function result | String | ⇒ | XML element reference (16 characters) |

## Description

The **DOM Get last child XML element** command returns an XML reference to the last “child” of the XML element passed as reference in *elementRef*. This reference may be used with the other XML parsing commands.

The optional *childElemName* and *childElemValue* parameters, when passed, receive respectively the name and value of the “child” element.

## Example

Recovery of the reference of the last XML element of the parent root. The XML structure (C:¥¥import.xml) is loaded into a BLOB beforehand:

```
C_BLOB(myBlobVar)
C_TEXT($ref_XML_Parent;$ref_XML_Child)
C_TEXT($childName;$childValue)

DOCUMENT TO BLOB("c:¥¥import.xml";myBlobVar)
$ref_XML_Parent:=DOM Parse XML variable(myBlobVar)
$ref_XML_Child:=DOM Get last child XML element($ref_XML_Parent;$childName;$childValue)
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

# DOM Get next sibling XML element

DOM Get next sibling XML element ( elementRef {; siblingElemName {; siblingElemValue}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| siblingElemName | String | ⇐ | Name of sibling XML element |
| siblingElemValue | String | ⇐ | Value of sibling XML element |
| Function result | String | ⇐ | Sibling XML element reference (16 characters) |

## Description

The **DOM Get next sibling XML element** command returns a reference to the next “sibling” of the XML element passed as reference. This reference can be used with other XML parsing commands.

The *siblingElemName* and *siblingElemValue* parameters, if they are passed, receive respectively the name and the value of the “sibling” element.

This command is used to navigate among the “children” of the XML element.

After the last “sibling,” the system variable OK is set to 0.

## Example 1

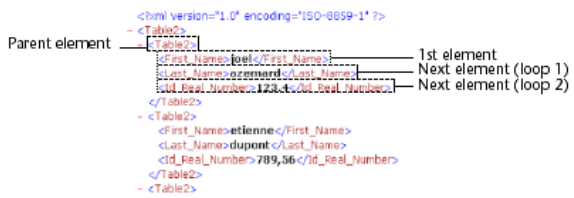
Retrieval of the reference of the next sibling XML element following the element passed as parameter:

```
C_TEXT($xml_Parent_Ref;$next_XML_Ref)
$next_XML_Ref:=DOM Get next sibling XML element($xml_Parent_Ref)
```


## Example 2

Retrieval in a reference loop of all the child XML elements following the parent element passed as parameter, beginning with the first child:

```
C_TEXT($xml_Parent_Ref;$first_XML_Ref;$next_XML_Ref)

$first_XML_Ref:=DOM Get first child XML element($xml_Parent_Ref)
$next_XML_Ref:=$first_XML_Ref
While(OK=1)
   $next_XML_Ref:=DOM Get next sibling XML element($next_XML_Ref)
End while
```




## System variables and sets

If the command has been correctly executed and if the parsed element is not the last “sibling” of the referenced element, the system variable OK is set to 1. If an error occurs or if the parsed element is the last “sibling” of the referenced element, it is set to 0.

# DOM Get parent XML element

DOM Get parent XML element ( elementRef {; parentElemName {; parentElemValue}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| parentElemName | String | ⇐ | Name of parent XML element |
| parentElemValue | String | ⇐ | Value of parent XML element |
| Function result | String | ⇒ | Parent XML element reference (16 characters) |

## Description

The **DOM Get parent XML element** command returns an XML reference to the “parent” of the XML element passed as reference in *elementRef*. This reference may be used with the other XML parsing commands.

The optional *parentElemName* and *parentElemValue* parameters, when passed, receive respectively the name and value of the parent element.

When you pass a root element in *elementRef*, the command returns the "#document" reference. The document node is the parent of a root element.
If you use this command on a document node, the command returns a null reference ("0000000000000000") and the OK variable is set to 0.

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

# DOM Get previous sibling XML element

DOM Get previous sibling XML element ( elementRef {; siblingElemName {; siblingElemValue}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| siblingElemName | String | ⇐ | Name of sibling XML element |
| siblingElemValue | String | ⇐ | Value of sibling XML element |
| Function result | String | ⇒ | Sibling XML element reference (16 characters) |

## Description

The **DOM Get previous sibling XML element** command returns a reference to the previous “sibling” of the XML element passed as reference. This reference may be used with the other XML parsing commands.

The optional *siblingElemName* and *siblingElemValue* parameters, when passed, receive respectively the name and value of the previous “sibling” element.

This command can be used to navigate among the “children” of an XML element.

Before the first “sibling,” the system variable OK is set to 0.

## System variables and sets

If the command has been executed correctly and if the referenced element is not the first “child” of the structure, the system variable OK is set to 1. If an error occurs or if the element parsed is the first “child” of the structure, it is set to 0.

# DOM Get Root XML element

DOM Get Root XML element ( elementRef ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| Function result | String | ⇐ | Reference of root element (16 characters) or "" in case of error |

## Description

The **DOM Get Root XML element** command returns a reference to the root element of the document to which the XML element that is passed in the *elementRef* parameter belongs. This reference can be used with the other XML parsing commands.

# DOM GET XML ATTRIBUTE BY INDEX

DOM GET XML ATTRIBUTE BY INDEX ( elementRef ; attribIndex ; attribName ; attribValue )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| attribIndex | Longint | ⇒ | Attribute index number |
| attribName | Variable | ⇐ | Attribute name |
| attribValue | Variable | ⇐ | Attribute value |

## Description

The **DOM GET XML ATTRIBUTE BY INDEX** command gets the name of an attribute specified by its index number as well as its value.

Pass the reference of an XML element in *elementRef* and the index number of the attribute that you want to know the name of in *attribIndex*. The name is returned in the *attribName* parameter and its value is returned in the *attribValue*, parameter. 4D attempts to convert the value obtained into the same type as that of the variable passed as parameter.

**Note:** The index number does not correspond to the location of the attribute in the XML file displayed in text form. In XML, the index of an attribute indicates it position among the attributes when placed in alphabetical order (based on their names). For an illustration of this, refer to the example of the **DOM Count XML attributes** command.

If the value passed in *attribIndex* is greater than the number of attributes present in the XML element, an error is returned.

## Example

Refer to the example in the **DOM Count XML attributes** command.

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. If an error occurs, it is set to 0.

# DOM GET XML ATTRIBUTE BY NAME

DOM GET XML ATTRIBUTE BY NAME ( elementRef ; attribName ; attribValue )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ➡ | XML element reference |
| attribName | String | ➡ | Attribute name |
| attribValue | Variable | ⬅ | Attribute value |

## Description

The **DOM GET XML ATTRIBUTE BY NAME** command gets the value of an attribute specified by name.

Pass the reference of an XML element in *elementRef* and the name of the attribute that you want to know the value of in *attribName*. The value is returned in the *attribValue* parameter. 4D attempts to convert the value obtained into the same type as that of the variable passed as parameter.

If no *attribName* attribute exists in the XML element, an error is returned. If several attributes of the XML element have the same name as that specified, only the value of the first attribute is returned.

## Example

This method is used to retrieve the value of an XML attribute using its name:

```
C_BLOB(myBlobVar)
C_TEXT($xml_Parent_Ref;$xml_Child_Ref)
C_LONGINT($LineNum)

$xml_Parent_Ref:=DOM Parse XML variable(myBlobVar)
$xml_Child_Ref:=DOM Get first child XML element($xml_Parent_Ref)
DOM GET XML ATTRIBUTE BY NAME($xml_Child_Ref;"N";$LineNum)
```

If this method is applied to the example below, $LineNum contains the value 1:

```
<?xml version="1.0" ?>
- <STANZA>
    <LINE N="1">I heard a thousand blended notes,</LINE>
    <LINE N="2">While in grove I sate reclined,</LINE>
    <LINE N="3">In that sweet mood when pleasant thoughts</LINE>
    <LINE N="4">Bring sad thoughts to the mind.</LINE>
  </STANZA>
```

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. If an error occurs, it is set to 0.

# DOM GET XML CHILD NODES

DOM GET XML CHILD NODES ( elementRef ; childTypesArr ; nodeRefsArr )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | Text | ⇒ | XML element reference |
| childTypesArr | Longint array | ⇐ | Types of child nodes |
| nodeRefsArr | Text array | ⇐ | References or Values of child nodes |

## Description

The **DOM GET XML CHILD NODES** command returns the types and references or values of all the child nodes of the XML element designated by *elementRef*.

The types of child nodes are returned in the *childTypesArr* array. You can compare the values returned by the command with the following constants, found in the "**XML**" theme:

| Constant | Type | Value |
|---|---|---|
| XML comment | Longint | 2 |
| XML processing instruction | Longint | 3 |
| XML DATA | Longint | 6 |
| XML CDATA | Longint | 7 |
| XML DOCTYPE | Longint | 10 |
| XML ELEMENT | Longint | 11 |

For more information, please refer to the description of the **DOM Append XML child node** command.

The *nodeRefsArr* array receives the values or references of the elements according to their nature (contents or instructions).

## Example

Given the following XML structure:

```
<myElement>Hello<br/>New<br/>York</myElement>
```

After executing these instructions:

```
elementRef:=DOM Find XML element($root;"myElement")
DOM GET XML CHILD NODES(elementRef;$typeArr;$textArr)
```

... the $typeArr and $textArr arrays will contain the following values:

```
$typeArr{1}=6    $textArr{1} = "Hello"
$typeArr{2}=11   $textArr{2} = "AEF1233456878977" (element reference <Br/>)
$typeArr{3}=6    $textArr{3} = "New"
$typeArr{4}=11   $textArr{4} = "AEF1237897734568" (element reference <Br/>)
$typeArr{5}=6    $textArr{5} = "York"
```

# DOM Get XML document ref

DOM Get XML document ref ( elementRef ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | Text | ⇒ | Reference of existing element in DOM tree |
| Function result | Text | ⇐ | Reference of first element of a DOM tree (document node) |

## Description

The **DOM Get XML document ref** command is used to recover the reference of the "document" element of the DOM tree whose reference you have passed in *elementRef*. The document element is the first element of a DOM tree; it is the parent of the root element.

The reference of the document element lets you handle the "Doctype" and "Processing Instruction" nodes. It can only be used with the **DOM Append XML child node** and **DOM GET XML CHILD NODES** commands.

At this level, you can only append processing instructions and comments or replace the *Doctype* node. You cannot create *CDATA* or *Text* nodes there.

## Example

In this example, we want to find the DTD declaration of the XML document:

```
C_TEXT($rootRef)
$rootRef:=DOM Parse XML source("")
If(OK=1)
   C_TEXT($documentRef)
 // we are looking for the document node, since it is the node to which
 // the DOCTYPE node is attached before the root node
   $documentRef:=DOM Get XML document ref($rootRef)
   ARRAY TEXT($typeArr;0)
   ARRAY TEXT($valueArr;0)
 // on this node we look for the DOCTYPE type node among the
 // child nodes
   DOM GET XML CHILD NODES($refDocument;$typeArr;$valueArr)
   C_TEXT($text)
   $text:=""
   $pos:=Find in array($typeArr;XML DOCTYPE)
   If($pos>-1)
 // We retrieve the DTD declaration in $text
      $text:=$text+"Doctype: "+$valueArr{$pos}+Char(Carriage return)
   End if
   DOM CLOSE XML($rootRef)
End if
```

# DOM Get XML element

DOM Get XML element ( elementRef ; elementName ; index ; elementValue ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ➡ | XML element reference |
| elementName | String | ➡ | Name of element to get |
| index | Longint | ➡ | Index number of element to get |
| elementValue | Variable | ⬅ | Value of the element |
| Function result | String | ↩ | XML reference (16 characters) |

## Description

The **DOM Get XML element** command returns a reference to the “child” element dependent on the *elementName* and *index* parameters.

The value of the element is also returned in the *elementValue* parameter.

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. If an error occurs, it is set to 0.

# DOM GET XML ELEMENT NAME

DOM GET XML ELEMENT NAME ( elementRef ; elementName )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| elementName | Variable | ⇐ | Name of the element |

## Description

The **DOM GET XML ELEMENT NAME** command returns, in the *elementName* parameter, the name of the XML element designated by *elementRef*. For more information on XML element names, refer to the **Overview of XML DOM Commands** section.

## Example

This method returns the name of the $xml_Element_Ref element:

```
C_TEXT($xml_Element_Ref)
C_TEXT($name)

DOM GET XML ELEMENT NAME($xml_Element_Ref;$name)
```

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. If an error occurs, it is set to 0.

# DOM GET XML ELEMENT VALUE

DOM GET XML ELEMENT VALUE ( elementRef ; elementValue {; cDATA} )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇨ | XML element reference |
| elementValue | Variable | ⇦ | Value of the element |
| cDATA | Variable | ⇦ | Contents of the CDATA section |

## Description

The **DOM GET XML ELEMENT VALUE** command returns, in the *elementValue* parameter, the value of the XML element designated by *elementRef*. 4D will attempt to convert the value obtained into the same type as that of the variable passed as parameter.

The optional *cDATA* parameter is used to retrieve the contents of the CDATA section(s) of the *elementRef* XML element. Like with the *elementValue* parameter, 4D will attempt to convert the value obtained into the same type as that of the variable passed as parameter.

**Note:** If the element designated by *elementRef* is a BLOB processed by the **DOM SET XML ELEMENT VALUE** command, it has been automatically encoded in base64. Therefore the command will automatically attempt to decode it in base64.

## Example

This method returns the value of the $xml_Element_Ref element:

```
C_TEXT($xml_Element_Ref)
C_REAL($value)

DOM GET XML ELEMENT VALUE($xml_Element_Ref;$value)
```

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. If an error occurs, it is set to 0.

# DOM Get XML information

DOM Get XML information ( elementRef ; xmlInfo ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇨ | XML root element reference |
| xmlInfo | Longint | ⇨ | Type of information to get |
| Function result | String | ⇦ | Value of the XML information |

## Description

The **DOM Get XML information** command retrieves diverse information about the XML element designated by *elementRef*.

In *xmlInfo*, pass a code indicating the type of information to be retrieved. You can use the following predefined constants, located in the **XML** theme:

| Constant | Type | Value | Comment |
|---|---|---|---|
| DOCTYPE Name | Longint | 3 | Name of the root element as defined in the DOCTYPE marker |
| Document URI | Longint | 6 | URI of the DTD |
| Encoding | Longint | 4 | Encoding used (UTF-8, ISO...) |
| PUBLIC ID | Longint | 1 | Public identifier (FPI) of the DTD to which the document conforms (if the DOCTYPE xxx PUBLIC tag is present). |
| SYSTEM ID | Longint | 2 | System identifier |
| Version | Longint | 5 | Accepted XML version |

# DOM Insert XML element

DOM Insert XML element ( targetElementRef ; sourceElementRef ; childIndex ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| targetElementRef | Text | ⇒ | Parent XML element reference |
| sourceElementRef | Text | ⇒ | XML element reference to insert |
| childIndex | Longint | ⇒ | Index of child of target element above which the new element must be inserted |
| Function result | Text | ⇐ | Reference of new XML element |

## Description

The **DOM Insert XML element** command can be used to insert a new XML element among the child elements of the XML element whose reference is passed in the *targetElementRef* parameter.

Pass the element to be inserted in *sourceElementRef*. This element must be passed as the reference of an existing XML element in a DOM tree.

The *childIndex* parameter can be used to designate the child of the parent element before which the new element must be inserted. Pass an index number in this parameter. If the value is not valid (for example, there is no child element having this index), the new element will be added before the first child of the parent element.

The command returns the reference of the XML element obtained.

## Example

In the following structure, we would like to invert the first and second book:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?> <BookCatalog>   <Book>           <Title>Open Source Web
Services</Title>           <Author>Collective</Author>           <Date>2003</Date>           <ISBN>2-7440-1507-5</ISBN>
      <Publisher>Wrox</Publisher>     </Book> <Book>           <Title>Building XML Web services</Title>
<Author>Scott Short</Author>           <Date>2002</Date>           <ISBN>2-10-006476-2</ISBN>
<Publisher>Microsoft Press</Publisher>     </Book> </BookCatalog>
```

To do this, simply execute the following code:

```
C_TEXT($rootRef)
$rootRef:=DOM Parse XML source("") //selection of XML document
If(OK=1)
   C_TEXT($newStruct)
   $newStruct:=DOM Create XML Ref("BookCatalog")

   $bookRef:=DOM Find XML element($rootRef;"/BookCatalog/Book[1]")
   $newElementRef:=DOM Append XML element($newStruct;$bookRef)

   $bookRef:=DOM Find XML element($rootRef;"/BookCatalog/Book[2]")
   C_TEXT($newElementRef)
   $newElementRef:=DOM Insert XML element($newStruct;$bookRef;1)

   DOM CLOSE XML($newStruct)
   DOM CLOSE XML($rootRef)
End if
```

# DOM Parse XML source

DOM Parse XML source ( document {; validation {; dtd | schema}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| document | String | ⇒ | Document pathname |
| validation | Boolean | ⇒ | True = Validation False = No validation |
| dtd \| schema | String | ⇒ | Location of the DTD or XML schema |
| Function result | String | ⇐ | Reference of XML element (16 characters) |

## Description

The **DOM Parse XML source** command parses a document containing an XML structure and returns a reference for this document. The command can validate (or not) the document via a DTD or an XML schema (XML Schema Definition (XSD) document).
The document can be located on the disk or on the Internet/Intranet.

**Note:** Execution of the **DOM Parse XML source** command is synchronous.

In the *document* parameter, you can pass:

- either a standard complete pathname (of the type C:¥¥Folder¥¥File¥¥... under Windows and MacintoshHD:Folder:File under Mac OS),
- or a Unix path under Mac OS (which must start with /).
- or a network path of the type http://www.site.com/File or ftp://public.ftp.com...

The Boolean parameter *validation* indicates whether or not to validate the structure.

- If *validation* equals True, the structure is validated. In this case, the parser attempts to validate the XML structure of the document based either on the DTD or XSD reference included in the document, or via the DTD or XML schema designated by the third parameter when it is passed.
- If *validation* equals False, the structure is not validated.

If you pass True in *validation* and omit the third parameter, the command attempts to validate the XML structure via a DTD or XSD reference found in the structure itself. Validation can be indirect: if the structure contains a reference to a DTD file that itself contains a reference to an XSD file, the command attempts to carry out both validations.

The third parameter indicates a specific DTD or an XML schema for document parsing. If you use this parameter, the command does not take the DTD referred to in the XML document into account.

**Validation by DTD**
There are two ways to specify a DTD:

- As a reference. To do this, pass the complete pathname of the new DTD (“dtd” extension) in the *dtd* parameter. If the document indicated does not contain a valid DTD, the *dtd* parameter is ignored and an error is generated.
- Directly in a text. In this case, if the contents of the parameter begin with “<?xml”, 4D will consider that it is the DTD; otherwise, 4D will consider it as a pathname.

**Validation by schema**
To validate the document via an XML schema, you just need to pass a file or URL with an “xsd” extension instead of a “dtd” one in the third parameter. Validation by XML schema is considered to be more flexible and more powerful than validation by DTD. The language of XSD documents is based on XML language. More particularly, XML schemas support data types. For more information about XML schemas, please refer to the following address: *http://www.w3.org/XML/Schema*.

If validation cannot be performed (no DTD or XSD, incorrect URL, etc.), an error is generated. The Error system variable indicates the error number. You can intercept this error using a method installed by the **ON ERR CALL** command.

The command returns a 16-character string (ElementRef) making up the reference in the memory of the document virtual structure. This reference should be used with other XML parsing commands.

**Important:** Once you no longer have any need for it, remember to call the **DOM CLOSE XML** command with this reference in order to free up the memory.

## Example 1

Opening an XML document located on disk, without validation:

```
$xml_Struct_Ref:=DOM Parse XML source("C:¥¥import.xml")
```

## Example 2

Opening an XML document located next to the database structure file, without validation:

```
$xml_Struct_Ref:=DOM Parse XML source("import.xml")
```

## Example 3

Opening an XML document located on disk and validation using a DTD on the disk:

```
$xml_Struct_Ref:=DOM Parse XML source("C:¥¥import.xml";True;"C:¥¥import_dtd.xml")
```

## Example 4

Opening an XML document located at a specific URL, without validation:

```
$xml_Struct_Ref:=DOM Parse XML source("http://www.4D.com/xml/import.xml")
```

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. Otherwise, it is set to 0.

# DOM Parse XML variable

DOM Parse XML variable ( variable {; validation {; dtd | schema}} ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| variable | BLOB, Text | ➡ | Name of the variable |
| validation | Boolean | ➡ | True = Validation by the DTD, False = No validation |
| dtd \| schema | String | ➡ | Location of the DTD or XML schema |
| Function result | String | ⬅ | Reference of XML element (16 characters) |

## Description

The **DOM Parse XML variable** command parses a BLOB or Text type variable containing an XML structure and returns a reference for this variable. The command can validate (or not) the structure via a DTD or an XML schema (XML Schema Definition (XSD) document).

Pass the name of the BLOB or Text variable containing the XML object in the *variable* parameter.

The Boolean parameter *validation* indicates whether or not to validate the structure.

- If *validation* equals True, the structure is validated. In this case, the parser attempts to validate the XML structure of the document based either on the DTD or XSD reference included in the document, or via the DTD or XML schema designated by the third parameter when it is passed.
- If *validation* equals False, the structure is not validated.

If you pass True in *validation* and omit the third parameter, the command attempts to validate the XML structure via a DTD or XSD reference found in the structure itself. Validation can be indirect: if the structure contains a reference to a DTD file that itself contains a reference to an XSD file, the command attempts to carry out both validations.

The third parameter indicates a specific DTD or an XML schema for document parsing. If you use this parameter, the command does not take the DTD referred to in the XML document into account.

**Validation by DTD**

There are two ways to specify a DTD:

- As a reference. To do this, pass the complete pathname of the new DTD (“dtd” extension) in the *dtd* parameter. If the document indicated does not contain a valid DTD, the *dtd* parameter is ignored and an error is generated.
- Directly in a text. In this case, if the contents of the parameter begin with “<?xml”, 4D will consider that it is the DTD; otherwise, 4D will consider it as a pathname.

**Validation by schema**

To validate the document via an XML schema, you just need to pass a file or URL with an “xsd” extension instead of a “dtd” one in the third parameter. Validation by XML schema is considered to be more flexible and more powerful than validation by DTD. The language of XSD documents is based on XML language. More particularly, XML schemas support data types. For more information about XML schemas, please refer to the following address: *http://www.w3.org/XML/Schema*.

If validation cannot be performed (no DTD or XSD, incorrect URL, etc.), an error is generated. The Error system variable indicates the error number. You can intercept this error using a method installed by the **ON ERR CALL** command.

The command returns a character string (*ElementRef*) making up the reference in the memory of the document virtual structure. This reference should be used with other XML parsing commands.

**Important:** Once you no longer have any need for it, remember to call the **DOM CLOSE XML** command with this reference in order to free up the memory.

## Example 1

Opening an XML object located in a 4D Text variable, without validation:

```
C_TEXT(myTextVar)
C_TIME(vDoc)
C_TEXT($xml_Struct_Ref)

vDoc:=Open document("Document.xml")
If(OK=1)
```

```
   RECEIVE PACKET(vDoc;myTextVar;32000)
   CLOSE DOCUMENT(vDoc)
   $xml_Struct_Ref:=DOM Parse XML variable(myTextVar)
End if
```

## Example 2

Opening an XML document located in a 4D BLOB, without validation:

```
C_BLOB(myBlobVar)
C_TEXT($ref_XML_Struct)

DOCUMENT TO BLOB(c¥¥import.xml;myBlobVar)
$xml_Struct_Ref:=DOM Parse XML variable(myBlobVar)
```

## System variables and sets

If the command has been correctly executed, the system variable OK is set to 1. Otherwise, it is set to 0.

# DOM REMOVE XML ATTRIBUTE

DOM REMOVE XML ATTRIBUTE ( elementRef ; attribName )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | Text | ⇒ | XML element reference |
| attribName | Text | ⇒ | Attribute to be removed |

## Description

The **DOM REMOVE XML ATTRIBUTE** command removes, if it exists, the attribute designated by *attribName* from the XML element whose reference is passed in the *elementRef* parameter.

If the attribute has been correctly removed, the *OK* system variable is set to 1. If no attribute named *attribName* exists in *elementRef*, an error is returned and the *OK* system variable is set to 0.

## Example

Given the following structure:

```
<?xml version="1.0" ?>
- <STANZA>
    <LINE N="1">I heard a thousand blended notes,</LINE>
    <LINE N="2">While in grove I sate reclined,</LINE>
    <LINE N="3">In that sweet mood when pleasant thoughts</LINE>
    <LINE N="4">Bring sad thoughts to the mind.</LINE>
  </STANZA>
```

The following code can be used to remove the first attribute "N=1":

```
C_BLOB(myBlobVar)
C_TEXT($xml_Parent_Ref;$xml_Child_Ref)
C_LONGINT($LineNum)

$xml_Parent_Ref:=DOM Parse XML variable(myBlobVar)
$xml_Child_Ref:=DOM Get first child XML element($xml_Parent_Ref)
DOM REMOVE XML ATTRIBUTE($xml_Child_Ref;"N")
```

# DOM REMOVE XML ELEMENT

DOM REMOVE XML ELEMENT ( elementRef )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇨ | XML element reference |

## Description

The **DOM REMOVE XML ELEMENT** command removes the element designated by *elementRef*.

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.
An error is generated when the element reference is invalid.

# DOM SET XML ATTRIBUTE

DOM SET XML ATTRIBUTE ( elementRef ; attribName ; attrValue {; attribName2 ; attrValue2 ; ... ; attribNameN ; attrValueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| attribName | String | ⇒ | Attribute to set |
| attrValue | String, Boolean, Longint, Real, Time, Date | ⇒ | New attribute value |

## Description

The **DOM SET XML ATTRIBUTE** command adds one or more attributes to the XML element whose reference is passed in the *elementRef* parameter. It also sets the value of each attribute defined.

Pass the attribute you want to set and its value respectively in the *attrName* and *attrValue* parameters (in the form of variables, fields or literal values). You can pass as many attribute/value pairs as you want.
The *attrValue* parameter can be of the text type or another type (Boolean, integer, real, date or time). If you pass a value other than text, 4D handles its conversion to text, according to the following principles:

| Type | Example of converted value |
|---|---|
| Boolean | "true" or "false" |
| Integer | "123456" |
| Real | "12.34" (the decimal separator is always ".") |
| Date | "2006-12-04T00:00:00Z" (RFC 3339 standard) |
| Time | "5233" (number of seconds) |

## Example

In the following XML source:

```
<Book>    <Title>The Best Seller</Title> </Book>
```

If the following code is executed:

```
vAttrName:="Font"
vAttrVal:="Verdana"
DOM SET XML ATTRIBUTE(vElemRef;vAttrName;vAttrVal)
```

We get:

```
<Book>    <Title Font=Verdana>The Best Seller</Title> </Book>
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# DOM SET XML DECLARATION

DOM SET XML DECLARATION ( elementRef ; encoding {; standalone {; indentation}} )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| encoding | String | ⇒ | XML document character set |
| standalone | Boolean | ⇒ | True = document is standalone False (default) = document is not standalone |
| indentation | Boolean | ⇒ | *** Obsolete, do not use *** |

## Description

The **DOM SET XML DECLARATION** command allows you to define various options that are useful in creating the XML tree set using *elementRef*. These options concern the encoding and the standalone property of the tree:

- *encoding*: Indicates the character set used in the document. By default (if the command is not called), the UTF-8 character set (compressed Unicode) is used.
  **Note:** If you pass a character set that is not supported by 4D XML commands, UTF-8 will be used. Refer to **Character Sets** to see the list of character sets supported (UTF-8 is however recommended in most cases).
- *standalone*: Indicates whether the tree is standalone (**True**) or if it needs other files or external resources to operate (**False**). By default (if the command is not called or if the parameter is omitted), the tree is not standalone.

**Compatibility note:** The *indentation* parameter is kept for reasons of compatibility with previous versions of 4D but its use is not recommended in 4D v12. From now on, to specify the indentation of the document, it is strongly recommended to use the **XML SET OPTIONS** command.

## Example

The following example sets the encoding to use and the standalone option in the *elementRef* element:

```
DOM SET XML DECLARATION(elementRef;"UTF-16";True)
```

# DOM SET XML ELEMENT NAME

DOM SET XML ELEMENT NAME ( elementRef ; elementName )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| elementName | String | ⇒ | New name of element |

## Description

The **DOM SET XML ELEMENT NAME** command modifies the name of the element set by *elementRef*.

Pass the reference of the element to rename in *elementRef* and the new name of the element in *elementName*. The command also takes charge of updating the open and close tags of the element.

## Example

In the following XML source:

```
<Book>    <Title>The Best Seller</Title> </Book>
```

If the following code is executed, with *vElemRef* containing the reference to the 'Book' element:

```
DOM SET XML ELEMENT NAME(vElemRef;"BestSeller")
```

We get:

```
<BestSeller>    <Title>The Best Seller</Title> </BestSeller>
```

## System variables and sets

If the command was executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

## Error management

An error is generated when:

- The element reference is invalid
- The new name of the element to create is invalid (for example, if it starts with a number).

# DOM SET XML ELEMENT VALUE

DOM SET XML ELEMENT VALUE ( elementRef {; xPath}; elementValue {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| elementRef | String | ⇒ | XML element reference |
| xPath | Text | ⇒ | XPath path of the XML element |
| elementValue | String, Variable | ⇒ | New value of element |
| * | Operator | ⇒ | If passed: set the value in CDATA |

## Description

The **DOM SET XML ELEMENT VALUE** command modifies the value of the element set by *elementRef*.

If you pass the optional *xPath* parameter, you choose to use XPath notation to indicate the element to be modified (for more information about this notation, refer to the section **Use of XPath notation (DOM)**). In this case, you must pass the reference of a root XML element in *elementRef* and the XPath path of the element to be modified in *xPath* .

In *elementValue*, pass a string or a variable (or a field) containing the new value of the specified element:

- If you pass a string, the value is used “as is” in the XML structure.
- If you pass a variable or a field, 4D processes the value, depending on the type of *elementValue*. All data types can be used, except arrays, pictures and pointers.

When the optional asterisk (*) parameter is passed, this indicates that the value of the element must be set in the form of CDATA. The special CDATA form can be used to write raw text as is (see example 2).

**Note:** If the element designated by *elementRef* is a BLOB processed by this command, it is automatically encoded in base64. In this case, the **DOM GET XML ELEMENT VALUE** command does automatically the reverse operation.

## Example 1

In the following XML source:

```
<Book>    <Title>The Best Seller</Title> </Book>
```

If the following code is executed, with *vElemRef* containing the reference to the “Title” element:

```
DOM SET XML ELEMENT VALUE(vElemRef;"The Loser")
```

We get:

```
<Book>    <Title>The Loser</Title> </Book>
```

## Example 2

In the following XML source:

```
<Maths>    <Postulate>1+2=3</Postulate> </Maths>
```

We want to write the text “12<18” in the *<Postulate>* element. This string cannot be written as is in XML because the “<” character is not accepted. This character must therefore be changed into “<” or the CDATA form must be used. If *vElemRef* indicates the XML *<Postulate>* node:

```
` Normal form
DOM SET XML ELEMENT VALUE(vElemRef;"12<18")
```

We get:

```
<Maths>    <Postulate>12 < 18</Postulate> </Maths>
```

```
` CDATA form
DOM SET XML ELEMENT VALUE(vElemRef;"12<18";*)
```

We get:

```
<Maths>    <Postulate><![CDATA[12 < 18]]></Postulate> </Maths>
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated (for example, if the element reference is invalid).

# XML SAX

- Overview of XML SAX Commands
- SAX ADD PROCESSING INSTRUCTION
- SAX ADD XML CDATA
- SAX ADD XML COMMENT
- SAX ADD XML DOCTYPE
- SAX ADD XML ELEMENT VALUE
- SAX CLOSE XML ELEMENT
- SAX GET XML CDATA
- SAX GET XML COMMENT
- SAX GET XML DOCUMENT VALUES
- SAX GET XML ELEMENT
- SAX GET XML ELEMENT VALUE
- SAX GET XML ENTITY
- SAX Get XML node
- SAX GET XML PROCESSING INSTRUCTION
- SAX OPEN XML ELEMENT
- SAX OPEN XML ELEMENT ARRAYS
- SAX SET XML DECLARATION

# Overview of XML SAX Commands

This theme groups together the XML SAX commands of 4D.

For general information about XML (overview, character sets, glossary) as well as the differences between the DOM and SAX modes, please refer to the **Overview of XML DOM Commands** section.

**Note concerning preemptive mode:** XML references created by a preemptive process can only be used in that specific process. Conversely, XML references created by a cooperative process can be used by any other cooperative process but cannot be used by any preemptive process.

## Creating, opening and closing XML documents via SAX

The SAX commands work with the standard document references of 4D (*DocRef*, Time type reference). It is therefore possible to use these commands jointly with the 4D commands used to manage documents, such as **SEND PACKET** or **Append document**.

The creation and opening of XML documents by programming is carried out using the **Create document** and **Open document** commands. Subsequently, the use of an XML command with these documents will cause the automatic implementation of XML mechanisms such as encoding. For instance, the *<?xml version="1.0" encoding="··· encodage ···" standalone = "no "?>* header will be written automatically in the document.

**Note:** Documents read by SAX commands must be opened in read-only mode by the **Open document** command. This avoids any conflict between 4D and the Xerces library when you open “standard” and XML documents simultaneously. If you execute a SAX parsing command with a document open in read-write mode, an alert message is displayed and parsing is impossible.

Closing an XML document must be carried out using the **CLOSE DOCUMENT** command. If any XML elements were open, they will be closed automatically.

# SAX ADD PROCESSING INSTRUCTION

SAX ADD PROCESSING INSTRUCTION ( document ; statement )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| statement | Text | ⇒ | Statement to insert in the document |

## Description

In the XML document referenced by *document*, the **SAX ADD PROCESSING INSTRUCTION** command adds an XML processing *statement*.

A processing statement lets you indicate the application type and when necessary any additional parameters allowing you to process an unparsable external entity.

The command formats the data of the statement in conformity with XML. However, the statements themselves are not parsed and it is up to the developer to make sure that they are valid.

## Example

The following code:

```
vtInstruct:="xml-stylesheet type="+Char(Quote)+"text/xsl"+Char(Quote)+
"href="+Char(Quote)+"style.xsl"+Char(Quote)
SAX ADD PROCESSING INSTRUCTION($DocRef;vtInstruct)
```

... will write the following line in the document:

```
<?xml-stylesheet type="text/xsl"href="style.xsl"?>
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# SAX ADD XML CDATA

SAX ADD XML CDATA ( document ; data )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| data | BLOB, Text | ⇒ | Text or BLOB to insert in the document between CData tags |

## Description

In the XML document referenced by *document*, the **SAX ADD XML CDATA** command adds *data* of the text or BLOB type. This *data* will be automatically framed between the *<![CDATA[* and *]]> tags.*

The text included in a CData section is ignored by the XML interpreter.

If you want to encode the contents of data, you must use the **BASE64 ENCODE** command. In this case, of course, you must pass a BLOB in *data*.

In order for this command to operate correctly, an element must be open. Otherwise, an error will be generated.

## Example

You want to insert the following lines in your XML document:

```
function matchwo(a,b) { if (a < b && a < 0) then       {       return 1       } else       {       return 0       } }
```

To do this, you just need to execute the following code:

```
C_TEXT(vtMytext)
... ` place the text in the vtMytext variable here
SAX ADD XML CDATA($DocRef;vtMytext)
```

The result will thus be:

```
<![CDATA[ function matchwo(a,b) { if (a < b && a < 0) then    {    return 1    } else    {    return 0    } } ]]>
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

# SAX ADD XML COMMENT

SAX ADD XML COMMENT ( document ; comment )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| comment | String | ⇒ | Comment to be added |

## Description

The **SAX ADD XML COMMENT** command adds a *comment* in the XML document referenced by *document*.

An XML comment is a text whose contents will not be parsed by the XML interpreter. XML comments must be enclosed between the <!-- and --> characters.

## Example

The following statement:

```
vComment:="Created by 4D"
SAX ADD XML COMMENT($DocRef;vComment)
```

... will write the following line in the document:

```
<!--Created by 4D-->
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

## Error management

In the event of an error, the command returns an error which can be intercepted using an error-handling method.

# SAX ADD XML DOCTYPE

SAX ADD XML DOCTYPE ( document ; docType )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| docType | String | ⇒ | DocType to be added |

## Description

The **SAX ADD XML DOCTYPE** command adds a DocType statement set by the *docType* parameter in the XML document referenced by *document*.

The DocType statement lets you indicate the type of XML in which the document has been written and to specify the Document Type Declaration (DTD) used. A DocType statement generally takes the following form: *<!DOCTYPE XML_type "DTD_address">*.

## Example

The following statement:

```
vDocType:="SYSTEM Books ¥"Book.DTD¥""
SAX ADD XML DOCTYPE($DocRef;vDocType)
```

... will write the following line in the document:

```
<!DOCTYPE SYSTEM Books"Book.DTD">
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

## Error management

In the event of an error, the the command returns an error which can be intercepted using an error-handling method.

# SAX ADD XML ELEMENT VALUE

SAX ADD XML ELEMENT VALUE ( document ; data {; *} )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| data | Text, Variable | ⇒ | Text or variable to insert in the document |
| * | Operator | ⇒ | If passed: encoding of special characters If omitted: no encoding |

## Description

In the XML document referenced by *document*, the **SAX ADD XML ELEMENT VALUE** command adds *data* directly without converting them. This command is equivalent, for instance, to inserting an attachment in the body of an e-mail.

In *data*, you can either pass a character string directly, or a 4D variable. The variable contents will be converted into text before being included in the XML document.
If you want to encode the contents of *data*, you must use the **BASE64 ENCODE** command. In this case, of course, you must passs a BLOB in *data*.

By default, the command encodes special characters (< > " '...) contained in the *data* parameter unless you have disabled this mechanism for the current process using the **XML SET OPTIONS** command by passing the XML raw data value to the XML string encoding option. For example:

```
XML SET OPTIONS($docRef;XML string encoding;XML raw data)
```

In this context, to force the encoding of special parameters when calling **SAX ADD XML ELEMENT VALUE**, you must pass the optional * parameter.

In order for this command to operate correctly, an element must be open. Otherwise, an error will be generated.

## Example

This example inserts the *whitepaper.pdf* file into the open XML element:

```
C_BLOB(vBMyBLOB)
DOCUMENT TO BLOB("c:¥¥whitepaper.pdf";vBMyBLOB)
SAX ADD XML ELEMENT VALUE($DocRef;vBMyBLOB)
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1; otherwise, it is set to 0 and an error is generated.

# SAX CLOSE XML ELEMENT

SAX CLOSE XML ELEMENT ( document )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ➡ | Reference of open document |

## Description

The **SAX CLOSE XML ELEMENT** command writes the statements necessary for closing the last element opened using the **SAX OPEN XML ELEMENT** command in the XML document referenced by *document*

The use of this command is optional. In fact, 4D will automatically add the necessary end tags for any unclosed elements when XML documents are closed.

## Example

If the last element opened is *<Book>*, the following statement:

```
SAX CLOSE XML ELEMENT($DocRef)
```

... will write the following line in the document:

```
</Book>
```

# SAX GET XML CDATA

SAX GET XML CDATA ( document ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ➡ | Reference of open document |
| value | Text, BLOB | ⬅ | Element value |

## Description

The **SAX GET XML CDATA** command gets the CDATA *value* of an XML element that exists in the XML document referenced in the *document* parameter. This command must be called with the XML CDATA SAX event. For more information about SAX events, refer to the description of the **SAX Get XML node** command.

Pass a *value* variable of the Text type if you want to retrieve data having a size greater than 32 KB (the database must be running in Unicode mode).

**Compatibility note:** Starting with 4D v12, CDATA contents encoded in base64 are automatically decoded by the **SAX GET XML CDATA** command, so it is not necessary to call the **BASE64 DECODE** command.

## Example

Let's look at the following piece of XML code:

```
<RootElement>    <Child>MyText<![CDATA[MyCData]]</Child> </RootElement>
```

The following 4D code will return “MyCData” in *vTextData*:

```
C_BLOB(vData)
C_TEXT(vTextData)
SAX GET XML CDATA(DocRef;vData)
vTextData:=BLOB to text(vData;UTF8 C string)
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# SAX GET XML COMMENT

SAX GET XML COMMENT ( document ; comment )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇨ | Reference of open document |
| comment | String | ⇦ | XML comment |

## Description

The **SAX GET XML COMMENT** command returns a *comment* if an XML Comment SAX event is generated in the XML document referenced in the *document* parameter. For more information about SAX events, refer to the description of the **SAX Get XML node** command.

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# SAX GET XML DOCUMENT VALUES

SAX GET XML DOCUMENT VALUES ( document ; encoding ; version ; standalone )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| encoding | String | ⇐ | XML document character set |
| version | String | ⇐ | XML version |
| standalone | Boolean | ⇐ | True = document is standalone, otherwise False |

## Description

The **SAX GET XML DOCUMENT VALUES** command gets basic information from the XML header of the XML document referenced in the *document* parameter.

The command returns the type of encoding, version and the “standalone” property of the document respectively in the *encoding*, *version* and *standalone* parameters. This command must be used with the SAX event XML Start Document. For more information about SAX events, refer to the description of the **SAX Get XML node** command.

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# SAX GET XML ELEMENT

SAX GET XML ELEMENT ( document ; name ; prefix ; attrNames ; attrValues )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| name | String | ⇐ | Element name |
| prefix | String | ⇐ | Namespace |
| attrNames | String array | ⇐ | Attribute names |
| attrValues | String array | ⇐ | Attribute values |

## Description

The **SAX GET XML ELEMENT** command returns various information about the element *name* that is present in the XML document reference in the *document* parameter. This command must be called with the XML Start Element or XML End Element SAX events. In the specific case of XML End Element, the attribute parameters are not handled. For more information about SAX events, refer to the description of the **SAX Get XML node** command.

The *name* parameter contains the name of the element.

The *prefix* parameter returns the namespace of the element. This parameter is empty if no namespace is linked to the element.

The command fills the *attrNames* array with the names of attributes of the target element. If necessary, the command creates and sizes the array automatically.

The command also fills the *attrValues* array with the values of attributes of the target element. If necessary, the command creates and sizes the array automatically.

## Example

Let's look at the following piece of XML code:

```
<RootElement>
<Child Att1="111"Att2="222"Att3="333">MyText</Child>
</RootElement>
```

Once the following statement has been executed:

```
SAX GET XML ELEMENT(DocRef;vName;vPrefix;tAttrNames;tAttrValues)
```

...*vName* will contain “Child”
*vPrefix* will contain “”
*tAttrNames{1}* will contain “Att1”, *tAttrNames{2}* will contain “Att2”, *tAttrNames{3}* will contain “Att3”
*tAttrValues{1}* will contain “111”, *tAttrValues{2}* will contain “222”, *tAttrValues{3}* will contain “333”

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# SAX GET XML ELEMENT VALUE

SAX GET XML ELEMENT VALUE ( document ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇨ | Reference of open document |
| value | Text, BLOB | ⇦ | Element value |

## Description

The **SAX GET XML ELEMENT VALUE** command allows you to get the *value* of an XML element that exists in the XML document referenced in the *document* parameter. This command must be called with the XML DATA SAX event. For more information about SAX events, refer to the description of the **SAX Get XML node** command.

Pass a Text or BLOB type variable in the *value* parameter. If you pass a BLOB, the command will automatically attempt to decode it into base64.

## Example

Let's look at the following piece of XML code:

```
<RootElement>    <Child Att1="111" Att2="222" Att3="333">MyText</Child> </RootElement>
```

The following instruction will return "MyText" in *vValue*:

```
SAX GET XML ELEMENT VALUE(DocRef;vValue)
```

## System variables and sets

If the command was executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# SAX GET XML ENTITY

SAX GET XML ENTITY ( document ; name ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| name | String | ⇐ | Entity name |
| value | String | ⇐ | Entity value |

## Description

The **SAX GET XML ENTITY** command allows you to get the *name* and *value* of an XML entity that exists in the XML document referenced in the *document* parameter. This command must be called with the XML Entity SAX event. For more information about SAX events, refer to the description of the **SAX Get XML node** command.

## Example

Let's look at the following piece of XML code:

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE body [    <!ELEMENT body (element*)>    <!ELEMENT element (#PCDATA)>
   <!ENTITY name "Replacement"> ]> <body>    <element>Entity updated by &name;</element> </body>
```

The following instruction will return “name” in *vName* and “Replacement” in *vValue*.

```
SAX GET XML ENTITY(DocRef;vName;vValue)
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# SAX Get XML node

SAX Get XML node ( document ) -> Function result

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ➡ | Reference of open document |
| Function result | Longint | ↩ | Event returned by function |

## Description

The **SAX Get XML node** command returns a long integer that indicates the type of SAX event returned while the XML document referenced in *document* is parsed.

Events that can be returned are available as "**XML**" theme constants:

| Constant | Type | Value |
|---|---|---|
| XML CDATA | Longint | 7 |
| XML Comment | Longint | 2 |
| XML DATA | Longint | 6 |
| XML End Document | Longint | 9 |
| XML End Element | Longint | 5 |
| XML Entity | Longint | 8 |
| XML Processing Instruction | Longint | 3 |
| XML Start Document | Longint | 1 |
| XML Start Element | Longint | 4 |

## Example

The following example processes an event:

```
DocRef:=Open document("";"xml";Read Mode)
If(OK=1)
   Repeat
      MyEvent:=SAX Get XML node(DocRef)
      Case of
         :(MyEvent=XML Start Document)
            DoSomething
         :(MyEvent=XML Comment)
            DoSomethingElse
      End case
   Until(MyEvent=XML End Document)
   CLOSE DOCUMENT(DocRef)
End if
```

## System variables and sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0 and an error is generated.

# SAX GET XML PROCESSING INSTRUCTION

SAX GET XML PROCESSING INSTRUCTION ( document ; name ; value )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| name | String | ⇐ | Instruction name |
| value | String | ⇐ | Instruction value |

## Description

The **SAX GET XML PROCESSING INSTRUCTION** command returns the *name* and *value* of the XML instruction processed in the XML document referenced in the *document* parameter. This command must be called with the XML Processing Instruction event. For more information about SAX events, refer to the description of the **SAX Get XML node** command.

## Example

Let's look at the following piece of XML code:

```
<?xml version="1.0" encoding="UTF-8"?> <!-- Edited with XML Spy v3.0.7 NT (http://www.xmlspy.com) by Myself (4D SA)--> <?PI TextProcess?> <!DOCTYPE RootElement SYSTEM "ParseTest.dtd">
```

The following instruction will return “PI” in *vName* and “TextProcess” in *vValue*:

```
SAX GET XML PROCESSING INSTRUCTION($DocRef;vName;vValue)
```

# SAX OPEN XML ELEMENT

SAX OPEN XML ELEMENT ( document ; tag {; attribName ; attribValue} {; attribName2 ; attribValue2 ; ... ; attribNameN ; attribValueN} )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| tag | String | ⇒ | Name of element to open |
| attribName | String | ⇒ | Attribute name |
| attribValue | String | ⇒ | Attribute value |

## Description

The **SAX OPEN XML ELEMENT** command adds a new element in the XML document referenced by *document* as well as, optionally, attributes and their values.

The added element is “open” in the document (the end tag is not added). To close an element created using this command, you must either:

- Use the **SAX CLOSE XML ELEMENT** command, or
- Close the XML document. In this case, 4D will automatically add the necessary XML end tags.

In *tag*, pass the name of the element to be created. This name may only contain letters, numbers and the characters “.”, “-”,”_” and “:”. If an invalid character is passed in *tag*, an error will be generated.

Optionally, the command can pass one or more attribute/value pairs (in the form of variables, fields or literal values) using the *attribName* and *attribValue* parameters. You can pass as many attribute/value pairs as you want.

## Example

The following statement:

```
vElement:="Book"
SAX OPEN XML ELEMENT($DocRef;vElement)
```

... writes the following line in the document:

```
<Book
```

## Error management

If an invalid character is passed in *tag*, an error is generated.

# SAX OPEN XML ELEMENT ARRAYS

SAX OPEN XML ELEMENT ARRAYS ( document ; tag {; attribNamesArray ; attribValuesArray} {; attribNamesArray2 ; attribValuesArray2 ; ... ; attribNamesArrayN ; attribValuesArrayN} )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| tag | String | ⇒ | Name of element to open |
| attribNamesArray | String array | ⇒ | Array of attribute names |
| attribValuesArray | String array, Longint array, Date array, Real array, Picture array, Boolean array | ⇒ | Array of attribute values |

## Description

The **SAX OPEN XML ELEMENT ARRAYS** command is used to add a new element in the XML document whose reference is passed in *document* as well as, optionally, attributes and their values in the form of arrays.

Except for the support of arrays (see below), this command is identical to **SAX OPEN XML ELEMENT**. Please refer to the description of this command for more information about its operation.

**SAX OPEN XML ELEMENT ARRAYS** accepts arrays of the date, number, Boolean and picture type as *attribValuesArray* parameter(s). 4D automatically carries out the necessary conversions; you can configure these conversions using the **XML SET OPTIONS** command.

Optionally, the **SAX OPEN XML ELEMENT ARRAYS** command can be used to pass pairs of attributes and attribute values in the form of arrays in the *attribNamesArray* and *attribValuesArray* parameters.

The arrays must have been created previously and operate in attribute/attribute value pairs. You can pass as many pairs of arrays, and as many items in each pair, as you want.

## Example

The following method:

```
ARRAY STRING(80;tAttrNames;2)
ARRAY STRING(80;tAttrValues;2)
vElement:="Book"
tAttrNames{1}:="Font"
tAttrValues{1}:="Arial"
tAttrNames{2}:="Style"
tAttrValues{2}:="Bold"
SAX OPEN XML ELEMENT ARRAYS($DocRef;vElement;tAttrNames;tAttrValues)
```

... will write in the document:

```
<Book Font="Arial" Style="Bold">
```

# SAX SET XML DECLARATION

SAX SET XML DECLARATION ( document ; encoding {; standalone {; indentation}} )

| Parameter | Type | | Description |
|---|---|---|---|
| document | DocRef | ⇒ | Reference of open document |
| encoding | String | ⇒ | XML document character set |
| standalone | Boolean | ⇒ | True = the document is standalone False (default) = document is not standalone |
| indentation | Boolean | ⇒ | *** Obsolete, do not use *** |

## Description

The **SAX SET XML DECLARATION** command initializes the XML document referenced in *document* using the values passed in the parameter. These parameters allow determining the encoding, standalone attribute and document indentation.

- *encoding*: Indicates the character set used in the document. By default (if the command is not called), the UTF-8 character set (compressed Unicode) is used.
  **Note:** If you pass a character set that is not supported by 4D XML commands, UTF-8 will be used. Refer to **Character Sets** to see the list of character sets supported (UTF-8 is however recommended in most cases).
- *standalone*: Indicates whether the document is standalone (**True**) or if it needs other files or external resources to operate (**False**). By default (if the command is not called or if the parameter is omitted), the document is not standalone.

**Compatibility note:** The *indentation* parameter is kept for reasons of compatibility with previous versions of 4D but its use is not recommended beginning with 4D v12. From now on, to specify the indentation of the document, it is strongly recommended to use the **XML SET OPTIONS** command.

This command must be called one time per document and before the first XML set command in the document; otherwise, an error message will be generated.

## Example

The following code:

```
SAX SET XML DECLARATION($DocRef;"UTF-16";True)
```

... will write this line in the document:

```
<<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
```

# List of constant themes

- 4D Environment
- 4D Write Pro
- ASCII Codes
- Backup and Restore
- BLOB
- Colors
- Communications
- Data File Maintenance
- Database Engine
- Database Events
- Database Parameters
- Date Display Formats
- Days and Months
- Design Object Access
- Dictionaries
- Digest Type
- Euro Currencies
- Events (Modifiers)
- Events (What)
- Expressions
- Field and Variable Types
- Find Window
- Font Styles
- Font Type List
- Form Area
- Form Events
- Form Object Types
- Form Objects (Access)
- Form Objects (Properties)
- Form Parameters
- Function Keys
- Graph Parameters
- Hierarchical Lists
- HTTP Client
- Index Type
- Is License Available
- ISO Latin Character Entities
- LDAP
- List Box
- Listbox Footer Calculation
- Log Events
- Math
- Menu Item Properties
- Multistyle Text
- Multistyle Text Attributes
- Open Form Window
- Open Window
- Pasteboard
- PHP
- Picture Compression
- Picture Display Formats
- Picture Metadata Names
- Picture Metadata Values
- Picture Transformation
- Platform Interface
- Platform Properties
- Print Options
- Process State

- Process Type
- QR Area Properties
- QR Borders
- QR Commands
- QR Document Properties
- QR Operators
- QR Output Destination
- QR Report Types
- QR Rows for Properties
- QR Text Properties
- Queries
- Relations
- Resources Properties
- SCREEN DEPTH
- SET RGB COLORS
- Shortcut and Associated Keys
- SQL
- Standard System Signatures
- System Documents
- System Folder
- System Format
- TCP Port Numbers
- Text Values for Associated Standard Action
- Time Display Formats
- Trigger Events
- Value for Associated Standard Action
- Web Area
- Web Server
- Web Services (Client)
- Web Services (Server)
- Windows
- XML

## 4D Environment

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_4D Interpreted desktop | Longint | 2 | |
| _o_Extras folder | Longint | 2 | |
| _o_Full Version | Longint | 0 | |
| 4D Client database folder | Longint | 3 | |
| 4D Desktop | Longint | 3 | |
| 4D Local mode | Longint | 0 | |
| 4D Remote mode | Longint | 4 | |
| 4D Server | Longint | 5 | |
| 4D Volume desktop | Longint | 1 | |
| 64 bit version | Longint | 1 | |
| Active 4D Folder | Longint | 0 | |
| Backup configuration file | Longint | 1 | Backup.xml file, stored in Preferences/Backup folder next to database structure file. |
| Current localization | Longint | 1 | Current language of the application: default language or language set via the **SET DATABASE LOCALIZATION** command. |
| Current resources folder | Longint | 6 | |
| Data folder | Longint | 9 | |
| Database folder | Longint | 4 | |
| Database folder Unix syntax | Longint | 5 | |
| Default localization | Longint | 0 | Language set automatically by 4D on startup according to the Resources folder and the system environment (not modifiable). |
| Demo version | Longint | 0 | |
| Full method text | Longint | 1 | |
| Highlighted method text | Longint | 2 | |
| HTML Root folder | Longint | 8 | |
| Internal 4D localization | Longint | 3 | Language used by 4D for sorts and text comparisons (set in the Preferences of the application). |
| Last backup file | Longint | 2 | Last backup file, named <databaseName>[bkpNum].4BK, stored at a custom location. |
| Licenses folder | Longint | 1 | |
| Logs folder | Longint | 7 | |
| Merged application | Longint | 2 | Version is an application merged with 4D Volume Desktop |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Structure settings | Longint | 0 | Access to "Structure settings" (default value if parameter omitted). In this mode, values used for *selector* are identical to those in standard mode. |
| User settings | Longint | 1 | Access to "User settings". In this mode, only certain keys can be used in the *selector* parameter |
| User settings file for data | Longint | 4 | settings.4DSettings file for current data file, stored in Preferences folder next to the data file. |
| User settings for data | Longint | 2 | Access to "User settings for data file", that is, user settings stored at the same level as the data file. In this mode, only certain keys can be used with the *selector* parameter (same subset as the User settings) |
| User structure settings file | Longint | 3 | settings.4DSettings file for all data files, stored in Preferences folder next to database structure file if enabled. |
| User system localization | Longint | 2 | Language set by the current user of the system. |

## 4D Write Pro

For more information, please refer to the **4D Write Pro Language** section.

| Constant | Type | Value | Comment |
|---|---|---|---|
| wk 4wp | Longint | 4 | The 4D Write Pro document is saved in a native archive format (zipped HTML and images saved in a separate folder). 4D specific tags are included and 4D expressions are not computed. This format is particularly suitable for saving and archiving 4D Write Pro documents on disk without any loss. |
| wk armenian | Longint | 19 | Traditional Armenian numbering style used (value for wk list style type) |
| wk author | String | author | **Used with:** Documents<br>**Default value:** none<br>**Possible values:** Custom string |
| wk auto | Longint | 0 | Value of property (constant) to which it is applied is adjusted automatically according to content or context of the element. |
| wk background clip | String | backgroundClip | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk border box<br>**Possible values:** wk border box / wk content box / wk padding box |
| wk background color | String | backgroundColor | **Used with:** Characters, Paragraphs, Documents, Pictures<br>**Default value:** wk transparent<br>**Description:** Background document is white by default(2) |
| wk background image | String | backgroundImage | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** none<br>**Description:** Image(3) |
| wk background origin | String | backgroundOrigin | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk padding box<br>**Possible values:** wk border box / wk content box / wk padding box |
| wk background position h | String | backgroundPositionH | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk left<br>**Possible values:** wk center / wk left / wk right |
| wk background position v | String | backgroundPositionV | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk top<br>**Possible values:** wk bottom / wk middle / wk top |
| wk background repeat | String | backgroundRepeat | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk repeat<br>**Possible values:** wk no repeat / wk repeat / wk repeat x / wk repeat y |
| wk background size h | String | backgroundSizeH | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk auto<br>**Possible values:** wk contain* / wk cover* / wk auto or Size(1)<br>*This value also affects wk background size v and modifies its value (CSS constraint). |
| wk background size v | String | backgroundSizeV | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk auto<br>**Possible values:** wk contain* / wk cover* / wk auto or Size(1)<br>*This value also affects wk background size h and modifies its value (CSS constraint). |
| wk bar | Longint | 4 | Inserts a vertical bar at tab position (value for wk tab stop types) |
| wk baseline | Longint | 4 | Aligns baseline of element with baseline of parent element (value for wk vertical align) |
| wk border box | Longint | 0 | Background is clipped to the border box (value for wk background clip) |
| wk border color | String | borderColor | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** black<br>**Description:** If there are multiple colors, **WP GET ATTRIBUTES** returns an empty string(2) |
| wk border color bottom | String | borderColorBottom | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** black<br>**Description:** Color(2) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| wk border color left | String | borderColorLeft | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** black<br>**Description:** Color(2) |
| wk border color right | String | borderColorRight | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** black<br>**Description:** Color(2) |
| wk border color top | String | borderColorTop | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** black<br>**Description:** Color(2) |
| wk border radius | String | borderRadius | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Integer(1), Radius in user unit |
| wk border style | String | borderStyle | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Possible values:** wk none / wk hidden / wk solid / wk dotted / wk dashed / wk double / wk groove / wk ridge / wk inset |
| wk border style bottom | String | borderStyleBottom | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Possible values:** wk none / wk hidden / wk solid / wk dotted / wk dashed / wk double / wk groove / wk ridge / wk inset |
| wk border style left | String | borderStyleLeft | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Possible values:** wk none / wk hidden / wk solid / wk dotted / wk dashed / wk double / wk groove / wk ridge / wk inset |
| wk border style right | String | borderStyleRight | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Possible values:** wk none / wk hidden / wk solid / wk dotted / wk dashed / wk double / wk groove / wk ridge / wk inset |
| wk border style top | String | borderStyleTop | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Possible values:** wk none / wk hidden / wk solid / wk dotted / wk dashed / wk double / wk groove / wk ridge / wk inset |
| wk border width | String | borderWidth | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** 2pt<br>**Description:** Integer(1) |
| wk border width bottom | String | borderWidthBottom | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** 2pt<br>**Description:** Integer(1) |
| wk border width left | String | borderWidthLeft | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** 2pt<br>**Description:** Integer(1) |
| wk border width right | String | borderWidthRight | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** 2pt<br>**Description:** Integer(1) |
| wk border width top | String | borderWidthTop | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** 2pt<br>**Description:** Integer(1) |
| wk bottom | Longint | 1 | Sets position of background image (value for wk background position v) or bottom of element aligned with lowest element on line (value for wk vertical align) |
| wk capitalize | Longint | 1 | Transforms first character of every word to uppercase (value for wk text transform) |
| wk center | Longint | 2 | Centers text or image (value for wk background position h, wk text align, and/or wk tab stop types) |
| wk circle | Longint | 11 | Circle-shaped glyph used (value for wk list style type) |
| wk cjk ideographic | Longint | 24 | Plain ideographic numbers used (value for wk list style type) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| wk club | Longint | 27 | Club-shaped glyph used (value for wk list style type) |
| wk company | String | company | **Used with:** Documents<br>**Default value:** none<br>**Description:** String |
| wk contain | Longint | -1 | Scales image to largest size such that its width and height fits inside content area (value for wk background size h and/or wk background size v) |
| wk content box | Longint | 2 | Background clipped to content box (value for wk background clip) or background image starts from upper left corner of content (value for wk background origin) |
| wk cover | Longint | -2 | Scales background image to largest size so that background area is completely covered by it (value for wk background size h and/or wk background size v) |
| wk custom | Longint | 29 | Custom marker used (value for wk list style type) |
| wk dashed | Longint | 3 | Dashed line used (value for wk text linethrough style and/or wk text underline style) |
| wk date creation | String | dateCreation | **Used with:** Documents<br>**Default value:** none<br>**Description:** Date *Value (dateCreation) is read-only and cannot be set |
| wk date modified | String | dateModified | **Used with:** Documents<br>**Default value:** none<br>**Description:** Date |
| wk decimal | Longint | 3 | Decimal alignment (value for wk tab stop types) or numbers used (value for wk list style type) |
| wk decimal greek | Longint | 28 | Greek numerals used (value for wk list style type) |
| wk decimal leading zero | Longint | 13 | Decimal numbers padded with initial zeros used (value for wk list style type) |
| wk default | Longint | -1 | Default value of property (constant) is used. |
| wk diamond | Longint | 26 | Diamond-shaped glyph used (value for wk list style type) |
| wk direction | String | direction | **Used with:** Paragraphs<br>**Default value:** wk left to right<br>**Possible values:** wk left to right / wk right to left |
| wk disc | Longint | 10 | Filled circle marker used (value for wk list style type) |
| wk dotted | Longint | 2 | Dotted line or border used (value for wk border style, wk text linethrough style and/or wk text underline style) |
| wk double | Longint | 4 | Double line or border used (value for wk border style, wk text linethrough style and/or wk text underline style) |
| wk dpi | String | dpi | **Used with:** Documents<br>**Default value:** none<br>**Description:** Integer, min: 72 - max: 1440 |
| wk end text | Longint | 0 | Sets end of document as end of text range |
| wk false | Longint | 0 | |
| wk font | String | font | **Used with:** Characters<br>**Default value:** Times<br>**Description:** Complete font name in string as given by the **FONT STYLE LIST** command. If the system does not recognize the font, it handles the substitution. If **WP SET ATTRIBUTES** does not recognize the font, it does nothing. |
| wk font bold | String | fontBold | **Used with:** Characters<br>**Default value:** wk false<br>**Description:** Abstract property which depends on the thickness of wk font. wk true if wk font has a font style whose weight is bold; wk false otherwise |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| wk font family | String | fontFamily | **Used with:** Characters<br>**Default value:** Times<br>**Description:** Font family as defined by wk font. Abstract property which modifies wk font. |
| wk font italic | String | fontItalic | **Used with:** Characters<br>**Default value:** wk false<br>**Description:** Abstract property which depends on the style of wk font. Value is wk true if wk font has a font style which is italic or oblique; wk false otherwise. |
| wk font size | String | fontSize | **Used with:** Characters<br>**Default value:** 12<br>**Description:** Size(1), only in points |
| wk georgian | Longint | 20 | Traditional Georgian numbering used (value for wk list style type) |
| wk groove | Longint | 6 | 3D grooved border used (value for wk border style) |
| wk hebrew | Longint | 21 | Traditional Hebrew numbering used (value for wk list style type) |
| wk height | String | height | **Used with:** Pictures<br>**Default value:** 0<br>**Description:** Size(1) or wk auto. Always automatic for document element (wk auto); the view handles the height. |
| wk hidden | Longint | 5 | No border used. Same as none (no border) except it takes precedence over all other conflicting borders (value for wk border style) |
| wk hiragana | Longint | 22 | Traditional Hiragana numbering used (value for wk list style type) |
| wk hollow square | Longint | 25 | Hollow square glyph used (value for wk list style type) |
| wk html debug | Longint | 1 | Formatted HTML code ("pretty print"), easier to debug |
| wk image | String | image | **Used with:** Pictures<br>**Default value:** none<br>**Description:** URL of image |
| wk image alternative text | String | imageAltText | **Used with:** Pictures<br>**Default value:** none<br>**Description:** HTML alternative text |
| wk inset | Longint | 8 | 3D inset border used (value for wk border style) |
| wk inside | String | Inside | **Used with:** Paragraphs<br>**Default value:** none<br>**Description:** For multi-paragraph ranges. To be added to a border, padding, or margin attribute to affect only the corresponding inter-paragraph property (not outside). |
| wk justify | Longint | 5 | Available for 4D Write Pro areas only |
| wk katakana | Longint | 23 | Traditional Katakana numbering used (value for wk list style type) |
| wk layout unit | String | userUnit | **Used with:** Documents<br>**Default value:** wk unit cm<br>**Possible values:** wk unit cm / wk unit pt / wk unit px / wk unit percent (only for line height and background size (h+v)) / wk unit mm / wk unit inch |
| wk left | Longint | 0 | Aligns text or tab to the left (value for wk text align or wk tab stop types) or sets starting position of background image (value for wk background position h) |
| wk left to right | Longint | 0 | Left-to-right text/writing direction used (value for wk direction) |
| wk line height | String | lineHeight | **Used with:** Paragraphs<br>**Default value:** wk normal<br>**Description:** Size(1) |
| wk linethrough | Longint | 2 | |

| Constant | Type | Value | Comment |
|---|---|---|---|
| wk list auto | Longint | 2147483647 | Restores/applies automatic list style values |
| wk list font | String | listFont | **Used with:** Paragraphs<br>**Default value:** Times<br>**Description:** Font name in string |
| wk list font family | String | listFontFamily | **Used with:** Paragraphs<br>**Default value:** Times<br>**Description:** Abstract property which depends on style of wk list font in string. |
| wk list start number | String | listStartNumber | **Used with:** Paragraphs<br>**Default value:** wk auto<br>**Possible values:** wk auto or an integer |
| wk list string format LTR | String | listStringFormatLtr | **Used with:** Paragraphs<br>**Default value:** # or string<br>**Description:** Bullet list direction if left-to-right (LTR) direction |
| wk list string format RTL | String | listStringFormatRtl | **Used with:** Paragraphs<br>**Default value:** # or string<br>**Description:** Bullet list direction if right-to-left (RTL) direction |
| wk list style image | String | listStyleImage | **Used with:** Paragraphs<br>**Default value:** wk none<br>**Description:** Image(3) |
| wk list style image height | String | listStyleImageHeight | **Used with:** Paragraphs<br>**Default value:** default<br>**Description:** Size(1) |
| wk list style type | String | listStyleType | **Used with:** Paragraphs<br>**Default value:** wk disc<br>**Description:** wk disc / wk circle / wk square / wk decimal(1 2 3) / wk decimal leading zero (01 02 03) / wk lower latin (a b c) / wk lower roman (i ii iii iv) / wk upper latin (A B C) / wk upper roman (I II III IV) / wk lower greek (alpha, beta, gamma, etc.) / wk armenian / wk georgian / wk hebrew / wk hiragana / wk katakana / wk cjk ideographic / wk hollow square / wk diamond / wk club / wk decimal greek / wk custom / wk none |
| wk lower greek | Longint | 18 | Lowercase classical Greek used (value for wk list style type) |
| wk lower latin | Longint | 14 | Lowercase ASCII letters used (value for wk list style type) |
| wk lower roman | Longint | 15 | Lowercase roman numerals used (value for wk list style type) |
| wk lowercase | Longint | 2 | Changes all characters to lowercase (value for wk text transform) |
| wk margin | String | margin | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk margin bottom | String | marginBottom | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk margin left | String | marginLeft | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk margin right | String | marginRight | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk margin top | String | marginTop | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk middle | Longint | 2 | Sets position of background image (value for wk background position v) or places element in middle of parent element (value for wk vertical align) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| wk mime html | Longint | 1 | 4D Write Pro document is saved as standard MIME HTML with HTML documents and images embedded as MIME parts (encoded in base64). Expressions are computed and 4D specific tags are removed. This format is particularly suitable for sending HTML emails with the **SMTP_QuickSend** command. |
| wk min height | String | minHeight | **Used with:** Paragraphs, Pictures<br>**Default value:** 0<br>**Description:** Size(1). Always automatic for document element; the view handles the height. |
| wk min width | String | minWidth | **Used with:** Paragraphs, Pictures<br>**Default value:** 0<br>**Description:** Size(1). Always automatic for document element; the view handles the width. |
| wk mixed | Longint | -2147483648 | |
| wk new line style sheet | String | newLineStyleSheet | **Used with:** Paragraphs<br>**Default value:** current style |
| wk no repeat | Longint | 3 | Background image will not be repeated (value for wk background repeat) |
| wk none | Longint | 0 | |
| wk normal | Longint | 0 | Standard HTML code |
| wk notes | String | notes | **Used with:** Documents<br>**Default value:** none<br>**Description:** String |
| wk outset | Longint | 9 | |
| wk outside | String | Outside | **Used with:** Paragraphs<br>**Default value:** none<br>**Description:** For multi-paragraph ranges. To be added to a border, padding, or margin attribute to affect only the corresponding outside paragraph property (not inside). |
| wk padding | String | padding | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk padding bottom | String | paddingBottom | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk padding box | Longint | 1 | Background clipped to padding box (value for wk background clip) or background image starts from upper left corner of padding edge (value for wk background origin) |
| wk padding left | String | paddingLeft | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk padding right | String | paddingRight | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk padding top | String | paddingTop | **Used with:** Paragraphs, Documents, Pictures<br>**Default value:** wk none<br>**Description:** Size(1) |
| wk range end | String | rangeEnd | (Read-only range attribute) |
| wk range owner | String | rangeOwner | (Read-only range attribute) |
| wk range start | String | rangeStart | (Read-only range attribute) |
| wk repeat | Longint | 0 | Background image repeated both vertically and horizontally (value for wk background repeat) |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| wk repeat x | Longint | 1 | Background image repeated only horizontally (value for wk background repeat) |
| wk repeat y | Longint | 2 | Background image repeated only vertically (value for wk background repeat) |
| wk ridge | Longint | 7 | 3D ridged border used (value for wk border style) |
| wk right | Longint | 1 | Aligns text or tab to the right (value for wk text align or wk tab stop types) or sets starting position of background image (value for wk background position h) |
| wk right to left | Longint | 1 | Right-to-left direction used (value for wk direction) |
| wk semi transparent | Longint | 5 | Semi-transparent line used (value for wk text linethrough style and/or wk text underline style) |
| wk small uppercase | Longint | 4 | Transforms all characters to small uppercase (value for wk text transform) |
| wk solid | Longint | 1 | Solid line or border used (value for wk border style, wk text linethrough style and/or wk text underline style) |
| wk square | Longint | 12 | Square marker used (value for wk list style type) |
| wk start text | Longint | 1 | Sets beginning of document as start of text range |
| wk style sheet | String | styleSheet | **Used with:** Paragraphs, Pictures<br>**Default value:** none<br>**Description:** Style sheet name in string |
| wk subject | String | subject | **Used with:** Documents<br>**Default value:** none<br>**Description:** String |
| wk subscript | Longint | 6 | Aligns element as subscript (value for wk vertical align) |
| wk superscript | Longint | 5 | Aligns element as superscript (value for wk vertical align) |
| wk tab stop offsets | String | tabStopOffsets | **Used with:** Paragraphs<br>**Default value:** 35.45pt<br>**Description:** Size(1) for array values in user unit. If not an array, SET will reset tab stops to a single default offset and GET will return last offset (which is the default relative offset for offsets beyond the last absolute offset). Max value: 10000pt(4) |
| wk tab stop types | String | tabStopTypes | **Used with:** Paragraphs<br>**Default value:** wk left<br>**Possible values:** wk left / wk right / wk center / wk decimal / wk bar or an array(4) |
| wk text align | String | textAlign | **Used with:** Paragraphs<br>**Default value:** wk left<br>**Possible values:** wk right / wk left / wk justify / wk center |
| wk text color | String | color | **Used with:** Characters<br>**Default value:** black<br>**Description:** Color(2) |
| wk text indent | String | textIndent | **Used with:** Paragraphs<br>**Default value:** 0<br>**Description:** Size(1) |
| wk text linethrough color | String | textLinethroughColor | **Used with:** Characters<br>**Default value:** textColor<br>**Description:** Color(2) |
| wk text linethrough style | String | textLinethroughStyle | **Used with:** Characters<br>**Default value:** wk none<br>**Possible values:** wk none / wk solid / wk dotted / wk dashed / wk double / wk semi transparent / wk word |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| wk text shadow color | String | textShadowColor | **Used with:** Characters<br>**Default value:** wk transparent<br>**Description:** Color(2) |
| wk text shadow offset | String | textShadowOffset | **Used with:** Characters<br>**Default value:** 1pt<br>**Description:** Size(1), only in points. |
| wk text transform | String | textTransform | **Used with:** Characters<br>**Default value:** wk none<br>**Description:** wk capitalize / wk lowercase / wk uppercase / wk small uppercase / wk none |
| wk text underline color | String | textUnderlineColor | **Used with:** Characters<br>**Default value:** textColor<br>**Description:** Color(2) |
| wk text underline style | String | textUnderlineStyle | **Used with:** Characters<br>**Default value:** wk none<br>**Possible values:** wk none / wk solid / wk dotted / wk dashed / wk double / wk semi transparent / wk word |
| wk title | String | title | **Used with:** Documents<br>**Default value:** New 4D Write Document<br>**Description:** String |
| wk top | Longint | 0 | Sets position of background image (value for wk background position v) or aligns element with top of tallest element on the line (value for wk vertical align) |
| wk transparent | Longint | -1 | Specifies color is transparent (value for wk background color or wk text shadow color) |
| wk true | Longint | 1 | |
| wk underline | Longint | 1 | |
| wk unit cm | String | cm | Unit used is centimeters (value for wk layout unit) |
| wk unit inch | String | in | Unit used is inches (value for wk layout unit) |
| wk unit mm | String | mm | Unit used is millimeters (value for wk layout unit) |
| wk unit percent | String | % | Unit used is a percentage (value for wk layout unit) |
| wk unit pt | String | pt | Unit used is points (value for wk layout unit) |
| wk unit px | String | px | Unit used is pixels (value for wk layout unit) |
| wk upper latin | Longint | 16 | Uppercase ASCII letters used (value for wk list style type) |
| wk upper roman | Longint | 17 | Uppercase roman numerals used (value for wk list style type) |
| wk uppercase | Longint | 3 | Changes all characters to uppercase (value for wk text transform) |
| wk value unit not percentage | Longint | -100000 | |
| wk value unit percentage | Longint | -100001 | |
| wk version | String | version | **Used with:** Documents<br>**Default value:** 1<br>**Description:** integer (from 1 to max). Get only; cannot be set. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| wk vertical align | String | verticalAlign | **Used with:** Characters, Paragraphs, Pictures<br>**Default value:** wk baseline<br>**Possible values:** wk top / wk bottom / wk middle / wk baseline / wk superscript / wk subscript. For characters, wk top and wk bottom have the same effect as wk baseline. For paragraphs, wk baseline, wk superscript and wk subscript have the same effect as wk top. |
| wk web page complete | Longint | 2 | .htm or .html extension. The document is saved as standard HTML and its resources are saved separately. 4D tags are removed and expressions are computed. This format is particularly suitable when you want to display a 4D Write Pro document in a web browser. |
| wk web page html 4D | Longint | 3 | 4D Write Pro document is saved as HTML and includes 4D specific tags; each expression is inserted as a non-breaking space. Since this format is lossless, it is appropriate for storing purposes in a text field. |
| wk width | String | width | **Default value:** 0<br>**Used with:** Paragraph, Picture<br>**Description:** Size(1) or wk auto. Always auto for document element; the view handles the width. |
| wk word | Longint | 6 | |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |

## ASCII Codes

| Constant | Type | Value | Comment |
|---|---|---|---|
| ACK ASCII code | Longint | 6 | |
| At sign | Longint | 64 | |
| Backspace | Longint | 8 | |
| BEL ASCII code | Longint | 7 | |
| BS ASCII code | Longint | 8 | |
| CAN ASCII code | Longint | 24 | |
| Carriage return | Longint | 13 | |
| CR ASCII code | Longint | 13 | |
| DC1 ASCII code | Longint | 17 | |
| DC2 ASCII code | Longint | 18 | |
| DC3 ASCII code | Longint | 19 | |
| DC4 ASCII code | Longint | 20 | |
| DEL ASCII code | Longint | 127 | |
| DLE ASCII code | Longint | 16 | |
| Double quote | Longint | 34 | |
| EM ASCII code | Longint | 25 | |
| ENQ ASCII code | Longint | 5 | |
| Enter | Longint | 3 | |
| EOT ASCII code | Longint | 4 | |
| ESC ASCII code | Longint | 27 | |
| Escape | Longint | 27 | |
| ETB ASCII code | Longint | 23 | |
| ETX ASCII code | Longint | 3 | |
| FF ASCII code | Longint | 12 | |
| FS ASCII code | Longint | 28 | |
| GS ASCII code | Longint | 29 | |
| HT ASCII code | Longint | 9 | |
| LF ASCII code | Longint | 10 | |
| Line feed | Longint | 10 | |
| NAK ASCII code | Longint | 21 | |
| NBSP | Longint | 202 | |
| NUL ASCII code | Longint | 0 | |
| Period | Longint | 46 | |
| Quote | Longint | 39 | |
| RS ASCII code | Longint | 30 | |
| SI ASCII code | Longint | 15 | |
| SO ASCII code | Longint | 14 | |
| SOH ASCII code | Longint | 1 | |
| SP ASCII code | Longint | 32 | |
| Space | Longint | 32 | |
| STX ASCII code | Longint | 2 | |
| SUB ASCII code | Longint | 26 | |
| SYN ASCII code | Longint | 22 | |
| Tab | Longint | 9 | |
| US ASCII code | Longint | 31 | |
| VT ASCII code | Longint | 11 | |

## Backup and Restore

| Constant | Type | Value | Comment |
|---|---|---|---|
| Auto repair mode | Longint | 1 | Use flexible mode with auto-repair actions and fill the *errObject* parameter (if any) |
| Field attribute with name | Longint | 2 | Fields are identified by their name. Example: {"LastName":"Jones"} |
| Field attribute with number | Longint | 1 | Fields are identified by their number (default if omitted). Example: {"5":"Jones"}. |
| Last backup date | Longint | 0 | |
| Last backup status | Longint | 2 | |
| Last restore date | Longint | 0 | |
| Last restore status | Longint | 2 | |
| Next backup date | Longint | 4 | |
| Strict mode | Longint | 0 | Use strict integration mode (default) |

## BLOB

| Constant | Type | Value | Comment |
|---|---|---|---|
| Compact compression mode | Longint | 1 | Compressed as much as possible (at the expense of the speed of compression and decompression operations). Default method. |
| Extended real format | Longint | 1 | |
| Fast compression mode | Longint | 2 | Compressed as fast as possible (and will be decompressed as fast as possible), at the expense of the compression ratio (the compressed BLOB will be bigger). |
| GZIP best compression mode | Longint | -1 | Most compact GZIP compression |
| GZIP fast compression mode | Longint | -2 | Fastest GZIP compression |
| Is not compressed | Longint | 0 | No compression |
| Mac C string | Longint | 0 | |
| Mac Pascal string | Longint | 1 | |
| Mac text with length | Longint | 2 | |
| Mac text without length | Longint | 3 | |
| Macintosh byte ordering | Longint | 1 | |
| Macintosh double real format | Longint | 2 | |
| Native byte ordering | Longint | 0 | |
| Native real format | Longint | 0 | |
| PC byte ordering | Longint | 2 | |
| PC double real format | Longint | 3 | |
| UTF8 C string | Longint | 4 | |
| UTF8 text with length | Longint | 5 | |
| UTF8 text without length | Longint | 6 | |

## Colors

| Constant | Type | Value | Comment |
|---|---|---|---|
| Black | Longint | 15 | |
| Blue | Longint | 6 | |
| Brown | Longint | 13 | |
| Dark blue | Longint | 5 | |
| Dark brown | Longint | 10 | |
| Dark green | Longint | 9 | |
| Dark grey | Longint | 11 | |
| Green | Longint | 8 | |
| Grey | Longint | 14 | |
| Light blue | Longint | 7 | |
| Light grey | Longint | 12 | |
| Orange | Longint | 2 | |
| Purple | Longint | 4 | |
| Red | Longint | 3 | |
| White | Longint | 0 | |
| Yellow | Longint | 1 | |

## Communications

| Constant | Type | Value | Comment |
|---|---|---|---|
| Data bits 5 | Longint | 0 | |
| Data bits 6 | Longint | 2048 | |
| Data bits 7 | Longint | 1024 | |
| Data bits 8 | Longint | 3072 | |
| MacOS printer port | Longint | 0 | |
| MacOS serial port | Longint | 1 | |
| Parity even | Longint | 12288 | |
| Parity none | Longint | 0 | |
| Parity odd | Longint | 4096 | |
| Protocol DTR | Longint | 30 | |
| Protocol none | Longint | 0 | |
| Protocol XONXOFF | Longint | 20 | |
| Speed 115200 | Longint | 1022 | |
| Speed 1200 | Longint | 94 | |
| Speed 1800 | Longint | 62 | |
| Speed 19200 | Longint | 4 | |
| Speed 230400 | Longint | 1021 | |
| Speed 2400 | Longint | 46 | |
| Speed 300 | Longint | 380 | |
| Speed 3600 | Longint | 30 | |
| Speed 4800 | Longint | 22 | |
| Speed 57600 | Longint | 0 | |
| Speed 600 | Longint | 189 | |
| Speed 7200 | Longint | 14 | |
| Speed 9600 | Longint | 10 | |
| Stop bits one | Longint | 16384 | |
| Stop bits one and a half | Longint | -32768 | |
| Stop bits two | Longint | -16384 | |

## Data File Maintenance

| Constant | Type | Value | Comment |
|---|---|---|---|
| Compact address table | Longint | 131072 | Force the address table of the records to be rewritten (slows down compacting). Note that in this case, record numbers are rewritten. If you only pass this option, 4D automatically enables the 'Update records' option. |
| Create process | Longint | 32768 | When this option is passed, compacting will be asynchronous and you will need to manage the results using the callback method (see below). 4D will not display the progress bar (it is possible to do so using the callback method). The OK system variable is set to 1 if the process has been launched correctly and 0 in all other cases. When this option is not passed, the OK variable is set to 1 if the compacting takes place correctly and 0 otherwise. |
| Do not compact index | Longint | 2 | |
| Do not create log file | Longint | 16384 | Generally, this command creates a log file in XML format (refer to the end of the command description). With this option, no log file will be created. |
| Move to replaced files folder | Longint | 4 | |
| New file | Longint | 0 | |
| New file dialog | Longint | 1 | |
| Renumber records | Longint | 1 | |
| Timestamp log file name | Longint | 262144 | When this option is passed, the name of the log file generated will contain the date and time of its creation; as a result, it will not replace any log file already generated previously. By default, if this option is not passed, log file names are not timestamped and each new file generated replaces the previous one. |
| Update records | Longint | 65536 | Force all records to be rewritten according to current definition of the fields in the structure |
| Use default folder | Longint | 1 | The data of the field passed as parameter are saved in the default folder, named *databaseName*.ExternalData and placed next to the data file. In this mode, external data are managed by 4D as if they were inside the data file. |
| Use selected file | Longint | 2 | |
| Use structure definition | Longint | 0 | 4D uses the parameters set in the structure for field storage (see the *Design Reference* manual). If you change from external storage to internal storage, the external file is not deleted. |
| Verify all | Longint | 16 | |
| Verify indexes | Longint | 8 | This option checks the physical consistency of the indexes, without any link with the data. It signals invalid keys but does not permit you to detect duplicated keys (two indexes that point to the same record). This type of error can only be detected with the Verify All option. |
| Verify records | Longint | 4 | |

## Database Engine

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| New record | Longint | -3 | |
| No current record | Longint | -1 | |

## Database Events

| Constant | Type | Value | Comment |
|---|---|---|---|
| On after host database exit | Longint | 4 | The **On Exit database method** of the host database has just finished running |
| On after host database startup | Longint | 2 | The **On Startup database method** of the host database just finished running |
| On application background move | Longint | 1 | The 4D application moves to the background |
| On application foreground move | Longint | 2 | The 4D application moves to the foreground |
| On before host database exit | Longint | 3 | The host database is closing. The **On Exit database method** of the host database has not yet been called.<br>The **On Exit database method** of the host database is not called while the **On Host Database Event database method** of the component is running |
| On before host database startup | Longint | 1 | The host database has just been started. The **On Startup database method** method of the host database has not yet been called.<br>The On Startup database method of the host database is not called while the **On Host Database Event database method** of the component is running |

## Database Parameters

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_Database cache size | Longint | 9 | **Scope**: 4D application<br>**Kept between two sessions**: -<br>**Description**: *Constant obsolete (kept for compatibility reasons only)*. We now recommend using the **Get cache size** command. |
| _o_Real display precision | Longint | 32 | **** *Selector disabled* **** |
| _o_Web conversion mode | Longint | 8 | **** *Selector disabled* **** |
| _o_Web Log recording | Longint | 29 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| 4D Local mode scheduler | Longint | 10 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Description**: see selector 12 |
| 4D Remote mode scheduler | Longint | 12 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Possible values**: for selectors 10, 11 and 12, the *value* parameter is expressed in hexadecimal *0x00aabbcc* as follows:<br>*aa* = minimum number of ticks per call to the system (0 to 100 included).<br>*bb* = maximum number of ticks per call to the system (0 to 100 included).<br>*cc* = number of ticks between calls to the system (0 to 20 included).<br>If one of the values is out of range, 4D sets it to its maximum. You can pass one of the following preset standard values in the *value* parameter:<br>• *value* = -1: maximum priority allocated to 4D,<br>• *value* = -2: average priority allocated to 4D,<br>• *value* = -3: minimum priority allocated to 4D.<br>**Description**: This parameter allows you to dynamically set the 4D system internal calls. Depending on the Selector, the scheduler value will be set for:<br>• 4D local mode when the command is called from a 4D single-user application (*selector=10*).<br>• 4D Server when the command is called from 4D Server (*selector=11*).<br>• 4D remote mode when the command is called from a 4D connected to 4D Server (*selector=12*).<br>**Note:** The operation of selector 12 (4D Remote Mode Scheduler) differs according to whether the **SET DATABASE PARAMETER** command is executed on the server machine or on the client machine:<br>- If the command is executed on the server machine, the new value will be applied to all the client machines that connect to it subsequently.<br>- If the command is executed on the client machine, the new value is applied to the client machine immediately as well as to all the client machines that connect to the server subsequently.<br>You can use this operation to implement a dynamic and individualized management of priority for each client machine. This consists in executing the command initially on the client machine to be configured, then a second time on the server machine using the default value, which will then be used for the client machines that connect to it subsequently.<br>This operation is in effect in 4D starting with versions 6.8.6, 2003.3 and 2004.<br>**Warning**: Configuring these selectors inappropriately can cause serious degradation of application performance. It is recommended to only modify the default values with full knowledge of the facts. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| 4D Remote mode timeout | Longint | 14 | **Scope** (legacy network layer only): 4D application if *value* positive<br>**Kept between two sessions**: Yes if *value* positive<br>**Description**: To be used in very specific cases. Value of the timeout granted by the remote 4D machine to the 4D Server machine. The default timeout value used by 4D in remote mode is set on the "Client-Server/Network options" page of the Database settings dialog box on the remote machine.<br>The 4D Remote mode timeout selector is only taken into account if you are using the legacy network. It is ignored when the *ServerNet* layer is activated: this setting is entirely managed by the 4D Server timeout (13) selector. |
| 4D Server log recording | Longint | 28 | **Scope**: 4D Server, 4D remote<br>**Kept between two sessions**: No<br>**Possible values**: 0 or from 1 to X (0 = do not record, 1 to X = sequential number, added to the file name).<br>**Description**: Starts or stops the recording of standard requests received by 4D Server (excluding Web requests). By default, the value is 0 (requests not recorded).<br>4D Server lets you record each request received by the server machine in a log file. When this mechanism is enabled, two files are created in the Logs folder of the database, next to the database structure file.They are named 4DRequestsLog_X.txt and 4DRequestsLog_ProcessInfo_X.txt, where X is the sequential number of the log. Once the file 4DRequestsLog has reached a size of 10 MB, it is closed and a new one is generated, with an incremented sequential number. If a file of the same name already exists, it is replaced directly. You can set the starting number of the sequence using the *value* parameter.<br>These text files store various information concerning each request in a simple tabbed format: time, process number, size of request, processing duration, etc. This information is particularly useful during the development phase of the application or for statistical purposes. It can be imported, for example, into a spreadsheet software in order to be processed. |
| 4D Server scheduler | Longint | 11 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Description**: see selector 12 |
| 4D Server timeout | Longint | 13 | **Scope**: 4D application if *value* positive<br>**Kept between two sessions**: Yes if *value* positive<br>**Possible values**: 0 -> 32 767<br>**Description**: Value of the 4D Server timeout. The default 4D Server timeout value is defined on the "Client-Server/Network options" page of the Database settings dialog box on the server side.<br>The server timeout sets the maximum period "authorized" to wait for a client response, for example when it is executing a blocking operation. After this period, 4D Server disconnects the client. The 4D Server Timeout selector allows you to set, in the corresponding *value* parameter, a new timeout expressed in minutes. This feature is particularly useful to increase the timeout before executing a blocking and time-consuming operation on the client, such as printing a large number of pages, which can cause an unexpected timeout. You also have two options:<br>• If you pass a **positive** value in the *value* parameter, you set a global and permanent timeout: the new value is applied to all process and is stored in the preferences of the 4D application (equivalent to change in the Preferences dialog box).<br>• If you pass a **negative** value in the *value* parameter, you set a local and temporary timeout: The new value is applied to the calling process only (the other processes keep the default values) and is reset to default as soon as the server receives any signal of activity from the client — for example, when the operation is finished. This option is useful for managing long operations initiated by 4D plug-ins.<br>To set the "No timeout" option, pass 0 in *value*. See example 1. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Auto synchro resources folder | Longint | 48 | **Scope**: 4D remote machine<br>**Kept between two sessions**: No<br>**Possible values**: 0 (no synchronization), 1 (auto synchronization) or 2 (ask).<br>**Description**: Dynamic synchronization mode for *Resources* folder of 4D client machine that executed the command with that of the server.<br>When the contents of the *Resources* folder on the server has been modified or a user has requested synchronization (for example via the resources explorer or following the execution of the **SET DATABASE LOCALIZATION** command), the server notifies the connected client machines.<br>Three synchronization modes are then possible on the client side. The Auto Synchro Resources Folder selector is used to specify the mode to be used by the client machine for the current session:<br>• 0 (default value): no dynamic synchronization (synchronization request is ignored)<br>• 1: automatic dynamic synchronization<br>• 2: display of a dialog box on the client machines, with the possibility of allowing or refusing synchronization.<br>The synchronization mode can also be set globally in the application Preferences. |
| Cache flush periodicity | Longint | 95 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: No<br>**Possible values**: longint > 1 (seconds)<br>**Description**: Gets or sets the current cache flush periodicity, expressed in seconds.<br>Modifying this value overrides the **Flush Cache every X Seconds** option in the **Database/Memory page** of the Database settings for the session (it is not stored in the Database settings). |
| Cache unload minimum size | Longint | 66 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Positive longint > 1.<br>**Description**: Minimum size of memory to release from the database cache when the engine needs to make space in order to allocate an object to it (value in bytes).<br>The purpose of this selector is to reduce the number of times that data is released from the cache in order to obtain better performance. You can vary this setting according to the size of the cache and that of the blocks of data being handled in your database.<br>By default, if this selector is not used, 4D unloads at least 10% of the cache when space is needed. |
| Character set | Longint | 17 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Circular log limitation | Longint | 90 | **Scope**: 4D local, 4D Server.<br>**Kept between two sessions**: No<br>**Possible values**: Any integer value, 0 = keep all logs<br>**Description**: Maximum number of files to keep in rotation for each type of log. By default, all files are kept. If you pass a value *X*, only the *X* most recent files are kept, with the oldest being erased automatically when a new one is created. This setting applies to each of the following log files: request logs (selectors 28 and 45), debug log (selector 34), events log (selector 79), as well as Web request logs and Web debug logs (selectors 29 and 84 of the **WEB SET OPTION** command). |
| Client character set | Longint | 24 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 17<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Client HTTPS port ID | Longint | 40 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 to 65535<br>**Description**: TCP port number used by the Web servers of the client machines for secure connections via SSL (HTTPS protocol). By default, the value is 443 (standard value).<br>This selector can be used to modify by programming the TCP port used by the Web servers of the client machines for secure connections via SSL (HTTPS protocol). By default, the value is 443 (standard value).<br>This selector operates exactly the same way as selector 39; however, it applies to all the 4D remote machines used as Web servers. If you only want to modify the value of certain specific client machines, use the Preferences dialog box of the remote 4D. |
| Client IP address to listen | Longint | 23 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 16<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client log recording | Longint | 45 | **Scope**: Remote 4D machine<br>**Kept between two sessions**: No<br>**Possible values**: 0 or from 1 to X (0 = do not record, 1 to X = sequential number, attached to file name).<br>**Description**: Starts or stops recording of standard requests carried out by the 4D client machine that executed the command (excluding Web requests). By default, the value is 0 (no recording of requests).<br>4D lets you record the log of requests carried out by the client machine. When this mechanism is activated, two files are created on the client machine, in the Logs subfolder of the local folder of the database. They are named 4DRequestsLog_X.txt and 4DRequestsLog_ProcessInfo_X.txt, where X is the sequential number of the log. Once the file 4DRequestsLog has reached a size of 10 MB, it is closed and a new one is generated, with an incremented sequential number. If a file with the same name already exists, it is directly replaced. You can set the starting number for the sequence using the value parameter.<br>These text files store various information concerning each request in a simple tabbed format: time, process number, size of request, processing duration, etc. This information is particularly useful during the development phase of the application or for statistical purposes |
| Client max concurrent Web proc | Longint | 25 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 18<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client Max Web requests size | Longint | 21 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 27<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client maximum Web process | Longint | 20 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 7<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Client minimum Web process | Longint | 19 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 6<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client port ID | Longint | 22 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: See selector 15<br>**Description**: Used to specify this parameter for all the remote 4D machines used as Web servers. The values defined using these selectors are applied to all the remote machines used as Web servers. If you want to define values only for certain remote machines, use the Preferences dialog box of 4D in remote mode. |
| Client Server port ID | Longint | 35 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 to 65535<br>**Description**: TCP port number where the 4D Server publishes the database (bound for 4D remote machines). By default, the value is 19813.<br>Customizing this value means that several 4D client-server applications can be used on the same machine with the TCP protocol; in this case, you must indicate a different port number for each application.<br>The value is stored in the database structure file. It can be set with 4D in local mode but is only taken into account in client-server configuration.<br>When you modify this value, it is necessary to restart the server machine in order for the new value to be taken into account. |
| Client Web log recording | Longint | 30 | **Scope**: All 4D remote machines<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 = Do not record (default), 1 = Record in CLF format, 2 = Record in DLF format, 3 = Record in ELF format, 4 = Record in WLF format.<br>**Description**: Starts or stops the recording of Web requests received by the Web servers of all the client machines. By default, the value is 0 (requests not recorded).<br>The operation of this selector is identical to that of selector 29; however, it applies to all the 4D remote machines used as Web servers. The "logweb.txt" file is, in this case, automatically placed in the Logs subfolder of the remote 4D database folder (cache folder).<br>If you only want to set values for certain client machines, use the Preferences dialog box of 4D in remote mode. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Debug log recording | Longint | 34 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Description**: Starts or stops the sequential recording of events occurring at the 4D programming level in the *4DDebugLog* file, which is automatically placed in the Logs subfolder of the database, next to the structure file. A new, more compact, tabbed text format is used in the event log file "4DDebugLog[_n].txt" starting with 4D v14 (where _n is the segment number of the file).<br>**Possible values**: Longint containing a bit field: value = bit1(1)+bit2(2)+bit3(4)+bit4(8)+…).<br>- Bit 1 (value 1) requests to enable the file (note that any other non-null value also enables it as well)<br>- Bit 2 (value 2) requests call parameters to methods and commands.<br>- Bit 3 (value 4) enables new tabbed format.<br>- Bit 4 (value 8) disables immediate writing of each operation on disk (enabled by default). Immediate writing is slower but more effective, for example for investigating causes of a crash. If you disable this mode, the file contents are more compact and are generated more quickly.<br>- Bit 5 (value 16) disables recording of plug-in calls (enabled by default).<br>In the (former) non-tabbed format, execution times are expressed in milliseconds and the "< ms" value is displayed when an operation lasts less than one millisecond.<br>In the new tabbed format, execution times are expressed in microseconds.<br>Examples:<br>SET DATABASE PARAMETER (34;1) // enables mode v13 file without parameters, with runtimes<br>SET DATABASE PARAMETER (34;2) // enables mode v13 file with parameters and runtimes<br>SET DATABASE PARAMETER (34;2+4) // enables file with v14 format, with parameters and runtimes<br>SET DATABASE PARAMETER (34;0) // disables file<br>To avoid having a file record too much information, you can restrict the 4D commands that are examined by using selector 80, Log Command list.<br>This option can be enabled for any type of 4D application (4D all modes, 4D Server, 4D Volume Desktop), in interpreted or compiled mode.<br>**Note:** This option is provided solely for the purpose of debugging and must not be put into production since it may lead to deterioration of the application performance and saturation of the hard disk. For more information about this format and on the use of the 4DDebugLog[_n].txt file, please contact the Technical Support of 4D Inc. |
| Diagnostic log recording | Longint | 79 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: 0 or 1 (0 = do not record, 1 = record)<br>**Description**: Starts or stops recording of the 4D diagnostic file. By default, the value is 0 (do not record).<br>4D can continuously record a set of events related to the internal application operation into a diagnostic file. Information contained in this file is intended for the development of 4D applications and can be analyzed with the help of the 4D tech support. When you pass 1 in this selector, a diagnostic file, named *DatabaseName_X*.txt, is automatically created (or opened) in the database **Logs** folder. Once this file reaches a size of 10 MB, it is closed and a new file named *DatabaseName_X*.txt is generated, with an incremented sequence number X.<br>Note that you can include custom information in this file using the **LOG EVENT** command. |
| Direct2D disabled | Longint | 0 | See selector 69 (Direct2D Status) |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Direct2D get active status | Longint | 74 | **Note**: You can only use this selector with the **Get database parameter** command and its value cannot be set.<br>**Description**: Returns active implementation of Direct2D under Windows.<br>**Possible values**: 0, 1, 2, 3, 4 or 5 (see values of selector 69). The value returned depends on the availability of Direct2D, the hardware and the quality of Direct2D support by the operating system.<br>For example, if you execute:<br>`SET DATABASE PARAMETER(Direct2D status;Direct2D Hardware)`<br>`$mode:=Get database parameter(Direct2D get active status)`<br><br>- On Windows 7 and higher, *$mode* is set to 1 when the system detects hardware compatible with Direct2D; otherwise, *$mode* is set to 3 (software context).<br>- On Windows Vista, *$mode* is set to 1 when the system detects hardware compatible with Direct2D; otherwise, *$mode* is set to 0 (disabling of Direct2D).<br>- On Windows XP, *$mode* is always set to 0 (not compatible with Direct2D). |
| Direct2D hardware | Longint | 1 | See selector 69 (Direct2D Status) |
| Direct2D software | Longint | 3 | See selector 69 (Direct2D Status) |
| Direct2D status | Longint | 69 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Description**: Activation mode to implement Direct2D under Windows.<br>**Possible values**: One of the following constants (mode 3 by default):<br>Direct2D Disabled (0): Direct2D mode is not enabled and the database functions in the previous mode (GDI/GDIPlus).<br>Direct2D Hardware (1): Use Direct2D as graphics hardware context for entire 4D application. If this context is not available, use Direct2D graphics software context (except under Vista, in which case GDI/GDIPlus mode is used for better performance).<br>Direct2D Software (3) (Default mode): Beginning with Windows 7, use Direct2D graphics software context for entire 4D application. Under Vista, GDI/GDIPlus mode is used for better performance.<br>**Compatibility note:** Starting with 4D v14, hybrid modes are disabled and redirected to available modes (the former mode 2 is equivalent to 1; former modes 4 and 5 are equivalent to mode 3). |
| HTTP compression level | Longint | 50 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| HTTP compression threshold | Longint | 51 | **Scope**:4D application<br>**Kept between two sessions**: No<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| HTTPS Port ID | Longint | 39 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Idle connections timeout | Longint | 54 | **Scope**:4D application unless value is negative<br>**Kept between two sessions**: No<br>**Possible values**: Whole value expressing a duration in seconds. The value can be positive (new connections) or negative (existing connections). By default, the value is 20.<br>**Description**: Maximum period of inactivity (timeout) for connections to both the 4D database engine and the SQL engine, as well as, in *ServerNet* mode (new network layer), to the 4D application server. When an idle connection reaches this limit, it is automatically put on standby, which freezes the client/server session and closes the network socket. In the server administration window, the state of the user process is indicated as "Postponed". This functioning is completely transparent for the user: as soon as there is new activity on the connection which is on standby, the socket is automatically reopened and the client/server session is restored.<br>On the one hand, this setting lets you save resources on the server: connections on standby close the socket and free up a process on the server. On the other hand, it lets you avoid losing connections due to the closing of idle sockets by the firewall. For this, the timeout value for idle connections must be lower than that of the firewall in this case.<br>If you pass a positive value in *value*, it applies to all new connections in all the processes. If you pass a negative value, it applies to connections that are open in the current process. If you pass 0, idle connections are not subjected to a timeout.<br>This parameter can be set on both the server and client side. If you pass two different durations, the shorter one is taken into account. Usually, you do not need to change this value. |
| Invert objects | Longint | 37 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0, 1 or 2 (0 = mode disabled, 1 = automatic mode, 2 = mode enabled).<br>**Description**: Configuration of the "object inversion" mode which is used to invert forms, objects, menu bars, and so on, in Application mode when the database is displayed under Windows in a right-to-left language. This mode can also be configured on the Interface/Right-to-left languages page of the Database Settings.<br>• Value 0 indicates that the mode is never enabled, regardless of the system configuration (corresponds to the Never value in the Database Settings).<br>• Value 1 indicates that the mode is enabled or disabled depending on the system configuration (corresponds to the Automatic value in the Database Settings).<br>• Value 2 indicates that the mode is enabled, regardless of the system configuration (corresponds to the Always value in the Database Settings).<br>For more information, refer to the *Design Reference* manual of 4D. |
| IP Address to listen | Longint | 16 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| JSON use local time | Longint | 85 | **Scope**: Current process<br>**Kept between two sessions**: No<br>**Possible values**: 0 = ignore local time zone, 1 (default) = take time zone into account.<br>**Description**: By default, 4D dates converted to JSON format take the local time zone into account. For example, converting the date !23/08/2013! gives you "2013-08-22T22:00:00Z" in JSON format when the operation is performed in France during Daylight Savings Time (GMT+2). This principle conforms to the standard operation of JavaScript. This can be a source of errors when you want to send JSON date values to someone in a different time zone. This is the case, for example, when you export a table using **Selection to JSON** in France that is meant to be reimported in the US using **JSON TO SELECTION**. By default, since dates are re-interpreted in each time zone, the values stored in the database will be different. In this case, you can modify the conversion mode for dates so that they do not take the time zone into account by passing 0 in this selector. Converting the date !23/08/2013! will then give you "2013-08-23T00:00:00Z" in all cases. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Log command list | String | 80 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: String containing a list of 4D command numbers to record (separated by semi-colons) or "all" to record all the commands or "" (empty string) to record none of them.<br>**Description**: List of 4D commands to record in the debugging file (see selector 34, Debug Log Recording). By default, all 4D commands are recorded.<br>This selector restricts the quantity of information saved in the debugging file by limiting the 4D commands whose execution you want to record. For example, you can write:<br>`SET DATABASE PARAMETER(Log command list;"277;341") //Record only the QUERY and QUERY SELECTION commands` |
| Max concurrent Web processes | Longint | 18 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |
| Maximum temporary memory size | Longint | 61 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Positive longint.<br>**Description**: Maximum size of temporary memory that 4D can allocate to each process, expressed in MB. By default, the value is 0 (no maximum size). 4D uses a special temporary memory dedicated to indexing and sorting operations. This memory is intended to preserve the "standard" cache memory during massive operations. It is activated only when needed. By default, the size of the temporary memory is limited only by the resources available (according to the system memory configuration).<br>This mechanism is suitable for most applications. However, in certain specific contexts, more particularly when a client-server application simultaneously carries out a large number of sequential sorts, the size of the temporary memory can increase critically, to the point where it can render the system unstable. In this context, setting a maximum size for the temporary memory allows you to preserve proper functioning of the application. In return, the running speed might be affected: when the maximum size is reached for a process, 4D uses disk files which may slow down processing. For specific needs such as those described above, a maximum size of around 50 MB is generally a good compromise. However, the ideal value will need to be determined according to the specificities of the application and will generally be the result of real-time volumetric testing. |
| Maximum Web process | Longint | 7 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 -> 32 767<br>**Description**: Maximum number of Web processes to maintain in non-contextual mode with 4D in local mode and 4D Server. By default, the value is 10.<br>In non-contextual mode, for the Web server to be reactive, 4D delays the Web processes for 5 seconds and reuses them to execute any possible future HTTP queries. In terms of performance, this is actually more advantageous than creating a new process for each query. Once a Web process is reused, it is delayed again for 5 seconds. When the maximum number of Web processes has been reached, the web process is then aborted. If no query has been attributed to a Web process within the 5 second delay, the process is aborted, except if the minimum number of Web processes has been reached (in which case the process is delayed again).<br>These parameters allow you to adjust how your Web server operates in relation to the number of requests and the memory available as well as other parameters. |
| Maximum Web requests size | Longint | 27 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: *Constant obsolete (kept for compatibility reasons only).* We now recommend using the **WEB SET OPTION** and **WEB GET OPTION** commands for configuring the HTTP server. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Minimum Web process | Longint | 6 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 -> 32 767<br>**Description**: Minimum number of Web processes to maintain in non-contextual mode with 4D in local mode and 4D Server. By default, the value is 0 (see below). |
| Number of formulas in cache | Longint | 92 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Positive longints<br>**Default value**: 0 (no cache)<br>**Description**: Sets or gets the maximum number of formulas to be kept in the cache of formulas, which is used by the **EXECUTE FORMULA** command. This limit is applied to all processes, but each process has its own formula cache. Caching formulas accelerates the **EXECUTE FORMULA** command execution in compiled mode since each cached formula is tokenized only once in this case. When you change the cache value, existing contents are reset even if the new size is larger than the previous one. Once the maximum number of formulas in the cache is reached, a new executed formula will erase the oldest one in the cache (FIFO mode). This parameter is only taken into account in compiled databases or compiled components. |
| Order by formula on server | Longint | 47 | **Scope**: Current table and process<br>**Kept between two sessions**: No<br>**Possible values**: 0 (use database configuration), 1 (execute on client) or 2 (execute on server)<br>**Description** : Execution location of **ORDER BY FORMULA** command for the table passed in the parameter.<br>When using a database in client-server mode, this command can be executed either on the server or on the client machine. This selector can be used to specify the execution location of this command (server or client). This mode can also be set in the database preferences. For more information, please refer to the description of selector 46, Query By Formula On Server.<br>**Note:** If you want to be able to enable "SQL type" joins (see the QUERY BY FORMULA Joins selector), you must always execute formulas on the server so that they have access to the records. Be careful, in this context, the formula must not contain any calls to a method, otherwise it will automatically be switched to the remote machine. |
| PHP interpreter IP address | Longint | 55 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Formatted string of the type "nnn.nnn.nnn.nnn" (for example "127.0.0.1").<br>**Description**:  IP address used locally by 4D to communicate with the PHP interpreter via FastCGI. By default, the value is "127.0.0.1". This address must correspond to the machine where 4D is located. This parameter can also be set globally for all the machines via the Database Settings.<br>For more information about the PHP interpreter, please refer to the *Design Reference* manual. |
| PHP interpreter port | Longint | 56 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Positive long integer type value. By default, the value is 8002.<br>**Description**: Number of the TCP port used by the PHP interpreter of 4D. This parameter can also be modified globally for all the machines via the Database Settings. For more information about the PHP interpreter, please refer to the *Design Reference* manual. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| PHP max requests | Longint | 58 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Positive long integer type value. By default, the value is 500.<br>**Description**: Maximum number of requests accepted by the PHP interpreter. When this maximum number is reached, the interpreter returns errors of the "server busy" type. For security or performance reasons, you can modify this value. This parameter can also be modified globally for all the machines via the Database Settings. For more information about this parameter, please refer to the FastCGI-PHP documentation.<br>**Note:** On the 4D side, these parameters are applied dynamically; it is not necessary to exit 4D in order for them to be taken into account. On the other hand, if the PHP interpreter is already launched, it will be necessary to exit and relaunch it in order for these modifications to be taken into account. |
| PHP number of children | Longint | 57 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values**: Positive long integer type value. By default, the value is 5.<br>**Description**: Number of child processes to be created and maintained locally by the PHP interpreter of 4D. For optimization reasons, the PHP interpreter creates and uses a set (pool) of system processes called "child processes" for processing script execution requests. You can vary the number of child processes according to the needs of your application. This parameter can also be modified globally for all the machines via the Database Settings. For more information about the PHP interpreter, please refer to the *Design Reference* manual.<br>**Note:** Under Mac OS, all the child processes share the same port. Under Windows, each child process uses a specific port number. The first number is the one set for the PHP interpreter; the other child processes increment the first one. For example, if the default port is 8002 and you launch 5 child processes, they will use ports 8002 to 8006. |
| PHP use external interpreter | Longint | 60 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Values** : 0 = use internal interpreter, 1 = use external interpreter<br>**Description**: Value indicating whether PHP requests in 4D are sent to the internal interpreter provided by 4D or to an external interpreter. By default the value is 0 (use of interpreter provided by 4D). If you want to use your own PHP interpreter, for example in order to use additional modules or a specific configuration, pass 1 in *value*. In this case, 4D does not launch its internal interpreter in the case of PHP requests.<br>The custom PHP interpreter must have been compiled in FastCGI and be located on the same machine as the 4D engine. Note that in this case, you must manage the interpreter entirely; it will not be started nor stopped by 4D. This parameter can also be modified globally for all the machines via the Database Settings. |
| Port ID | Longint | 15 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: No<br>**Description**: TCP port ID used by the 4D Web server with 4D in local mode and 4D Server. The default value, which can be set on the "Web/Configuration" page of the Preferences dialog box, is 80. You can use the constants of the **TCP Port Numbers** theme for the *value* parameter.<br>The Port ID selector is useful for 4D Web Servers compiled and merged with 4D Desktop (in which there is no access to the Design mode). For more information about the TCP port ID, refer to the **Web Server Settings** section. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Query by formula joins | Longint | 49 | **Scope**: Current process<br>**Kept between two sessions**: No<br>**Possible values**: 0 (use database configuration), 1 (always use automatic relations) or 2 (use SQL joins if possible).<br>**Description**: Operating mode of the **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** commands relating to the use of "SQL joins."<br>In databases created starting with version 11.2 of 4D v11 SQL, these commands carry out joins based on the SQL joins model. This mechanism can be used to modify the selection of a table according to a query carried out on another table without these tables being connected by an automatic relation (necessary condition in previous versions of 4D).<br>The QUERY BY FORMULA Joins selector lets you specify the operating mode of the query by formula commands for the current process:<br>• 0: Uses the current settings of the database (default value). In databases created starting with version 11.2 of 4D v11 SQL, "SQL joins" are always activated for queries by formula. In converted databases, this mechanism is not activated by default for compatibility reasons but can be implemented via a preference.<br>• 1: Always use automatic relations (= functioning of previous versions of 4D). In this mode, a relation is necessary in order to set the selection of a table according to queries carried out on another table. 4D does not do "SQL joins."<br>• 2: Use SQL joins if possible (= default operation of databases created in version 11.2 and higher of 4D v11 SQL ). In this mode, 4D establishes "SQL joins" for queries by formula when the formula is suited for it (with two notable exceptions, see the description of the **QUERY BY FORMULA** or **QUERY SELECTION BY FORMULA** command).<br>**Note:** With 4D in remote mode, "SQL joins" can only be used if the formulas are executed on the server (they must have access to the records). To configure where formulas are to be executed, please refer to selectors 46 and 47. |
| Query by formula on server | Longint | 46 | **Scope**: Current table and process<br>**Kept between two sessions**: No<br>**Possible values**: 0 (use database configuration), 1 (execute on client) or 2 (execute on server)<br>**Description**: Execution location of **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** commands for the *table* passed in the parameter.<br>When using a database in client-server mode, the query "by formula" commands can be executed either on the server or on the client machine:<br>• In databases created with 4D v11 SQL, these commands are executed on the server.<br>• In converted databases, these commands are executed on the client machine, as in previous versions of 4D.<br>• In converted databases, a specific preference (Application/Compatibility page) can be used to globally modify the execution location of these commands.<br>This difference in execution location influences not only application performance (execution on the server is usually faster) but also programming. In fact, the value of the components of the formula (in particular variables called via a method) differ according to the execution context. You can use this selector to punctually adapt the operation of your application.<br>If you pass 0 in the *value* l'parameter, the execution location of query "by formula" commands will depend on the database configuration: in databases created with 4D v11 SQL, these commands will be executed on the server. In converted databases, they will be executed on the client machine or the server according to the database preferences. Pass 1 or 2 in *value* to "force" the execution of these commands, respectively, on the client or on the server machine.<br>Refer to example 4.<br>**Note:** If you want to be able to enable "SQL type" joins (see the QUERY BY FORMULA Joins selector), you must always execute formulas on the server so that they have access to the records. Be careful, in this context, the formula must not contain any calls to a method, otherwise it will automatically be switched to the remote machine. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| QuickTime support | Longint | 82 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (default) = QuickTime disabled, 1 = QuickTime enabled.<br>**Description**: In 4D starting with v14, by default QuickTime codecs are no longer supported. For compatibility, you can use this selector to re-enable them in your database. Modification of this option requires that the database be restarted. Nevertheless, you should note that in future versions of 4D, QuickTime support is permanently removed. |
| Server base process stack size | Longint | 53 | **Scope**: 4D Server<br>**Kept between two sessions**: No<br>**Possible values**: Positive longint.<br>**Description**: Size of the stack allocated to each preemptive system process on the server, expressed in bytes. The default size is determined by the system.<br>Preemptive system processes (processes of the 4D client base process type) are loaded to control the main 4D client processes. The size allocated by default to the stack of each preemptive process allows a good ease of execution but may prove to be consequential when very large numbers of processes (several hundred) are created.<br>For optimization purposes, this size can be reduced considerably if the operations carried out by the database allow for it (for example if the database does not carry out sorts of large quantities of records). Values of 512 or even 256 KB are possible. Be careful, under-sizing the stack is critical and can be harmful to the operation of 4D Server. Setting this parameter should be done with caution and must take the database conditions of use into account (number of records, type of operations, etc.).<br>In order to be taken into account, this parameter must be executed on the server machine (for example in the **On Server Startup Database Method**). |
| Spellchecker | Longint | 81 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: 0 (default) = native OS X spellchecker (Hunspell disabled), 1 = Hunspell spellcheck enabled.<br>**Description**: Enables the Hunspell spellcheck under OS X. By default, the native spellchecker is enabled on this platform. You may prefer to use the Hunspell spellcheck, for example, in order to unify the interface for your cross-platform applications (under Windows, only the Hunspell spellcheck is available). For more information, refer to **Support of Hunspell dictionaries**. |
| SQL Autocommit | Longint | 43 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (deactivation) or 1 (activation)<br>**Description**: Activation or deactivation of the SQL auto-commit mode. By default, the value is 0 (deactivated mode)<br>The auto-commit mode is used to strengthen the referential integrity of the database. When this mode is active, all **SELECT**, **INSERT**, **UPDATE** and **DELETE** (SIUD) queries are automatically included in ad hoc transactions when they are not already executed within a transaction. This mode can also be set in the Preferences of the database. |
| SQL Engine case sensitivity | Longint | 44 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (case not taken into account) or 1 (case-sensitive)<br>**Description**: Activation or deactivation of case-sensitivity for string comparisons carried out by the SQL engine.<br>By default, the value is 1 (case-sensitive): the SQL engine differentiates between upper and lower case and between accented characters when comparing strings (sorts and queries). For example “ABC”= “ABC” but “ABC” # “Abc” and "abc" # "âbc." In certain cases, for example so as to align the functioning of the SQL engine with that of the 4D engine, you may wish for string comparisons to not be case-sensitive (“ABC”=“Abc"="âbc").<br>This option can also be set on the **SQL page** of the Database settings. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| SQL Server Port ID | Longint | 88 | **Scope**: 4D local, 4D Server.<br>**Kept between two sessions**: Yes<br>**Description**: Gets or sets the TCP port number used by the integrated SQL server of 4D in local mode or 4D Server. By default, the value is 19812. When this selector is set, the database setting is updated. You can also set the TCP port number on the "SQL" page of the Database Settings dialog box.<br>**Possible values**: 0 to 65535.<br>**Default value**: 19812 |
| SSL cipher list | String | 64 | **Scope**: 4D application<br>**Kept between two sessions**: No<br>**Possible values**: Sequence of strings separated by colons (for example "RC4-MD5:RC4-64-MD5:....")<br>**Description:** Cipher list used by 4D for the secure protocol. This list modifies the priority of ciphering algorithms implemented by 4D. For example, you can pass the following string in the *value* parameter: "AES:ALL:!aNULL:!eNULL:+RC4:@STRENGTH". For a complete description of the syntax for the ciphers list, refer to the ciphers page of the OpenSSL site. This setting applies to the entire 4D application (it concerns the HTTP server, SQL server, client/server connections, as well as the HTTP client and all the 4D commands that make use of the secure protocol) but it is temporary (it is not maintained between sessions). When the cipher list has been modified, you will need to restart the server concerned in order for the new settings to be taken into account.<br>To reset the cipher list to its default value (stored permanently in the SLI file), call the **SET DATABASE PARAMETER** command and pass an empty string ("") in the *value* parameter.<br>**Note:** With the **Get database parameter** command, the cipher list is returned in the optional *stringValue* parameter and the return parameter is always 0. |
| Table sequence number | Longint | 31 | **Scope**: 4D application<br>**Kept between two sessions**: Yes<br>**Possible values**: Any longint value.<br>**Description**: This selector is used to modify or get the current unique number for records of the table passed as parameter. "Current number" means "last number used": if you modify this value using **SET DATABASE PARAMETER**, the next record will be created with a number that consists of the value passed + 1. This new number is the one returned by the **Sequence number** command as well in any field of the table to which the "Autoincrement" property has been assigned in the Structure editor or via SQL.<br>By default, this unique number is set by 4D and corresponds to the order of record creation. For additional information, refer to the documentation of the **Sequence number** command. |
| Unicode mode | Longint | 41 | **Scope**: Database<br>**Kept between two sessions**: Yes<br>**Possible values**: 0 (compatibility mode) or 1 (Unicode mode)<br>**Description**: Current database operating mode, with regards to the character set. 4D supports the Unicode character set but can function in “compatibility” mode (based on the Mac ASCII character set). By default, converted databases are executed in compatibility mode (0) and databases created with version 11 or higher are executed in Unicode mode. The execution mode can be controlled via an option in the Preferences and can also be read or (for testing purposes) modified via this selector. Modifying this option requires the database to be restarted in order for it to be taken into account. Note that within a component it is not possible to modify this value, but only to read it. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Use legacy network layer | Longint | 87 | **Scope**: 4D in local mode, 4D Server<br>**Kept between two sessions**: Yes<br>**Description:** Sets or gets the current status of the legacy network layer for client/server connections. The legacy network layer is obsolete beginning with 4D v14 R5 and should be replaced progressively in your applications with the *ServerNet* network layer. *ServerNet* will be required in upcoming 4D releases in order to benefit from future network evolutions. For compatibility reasons, the legacy network layer is still supported to allow a smooth transition for existing applications; (it is used by default in applications converted from a release prior to v14 R5). Pass 1 in this parameter to use the legacy network layer (and disable *ServerNet*) for your client/server connections, and pass 0 to disable the legacy network (and use the *ServerNet*).<br>This property can also be set by means of the "Use legacy network layer" option found on the **Compatibility page** of the Database Settings (see **Network and Client-Server options**). In this section, you will also find a discussion about migration strategy. We recommend that you activate the *ServerNet* as soon as possible.<br>You will need to restart the application in order for this parameter to be taken into account. It is not available in 4D Server v14 R5 64-bit version for OS X, which only supports the *ServetNet*; (it always returns 0).<br>**Possible values:** 0 or 1 (0 = do not use legacy layer, 1 = use legacy layer)<br>**Default value:** 0 in databases created with 4D v14 R5 or higher, 1 in databases converted from 4D v14 R4 or earlier. |

## Date Display Formats

| Constant | Type | Value | Comment |
|---|---|---|---|
| Blank if null date | Longint | 100 | "" instead of 0 |
| Date RFC 1123 | Longint | 10 | Fri, 10 Sep 2010 13:07:20 GMT |
| Internal date abbreviated | Longint | 6 | Dec 29, 2006 |
| Internal date long | Longint | 5 | December 29, 2006 |
| Internal date short | Longint | 7 | 12/29/2006 |
| Internal date short special | Longint | 4 | 12/29/06 (but 12/29/1896 or 12/29/2096) |
| ISO Date | Longint | 8 | 2006-12-29T00:00:00 (deprecated) |
| ISO Date GMT | Longint | 9 | 2010-09-13T16:11:53Z |
| System date abbreviated | Longint | 2 | Sun, Dec 29, 2006 |
| System date long | Longint | 3 | Sunday, December 29, 2006 |
| System date short | Longint | 1 | 12/29/2006 |

## Days and Months

| Constant | Type | Value | Comment |
|---|---|---|---|
| April | Longint | 4 | |
| August | Longint | 8 | |
| December | Longint | 12 | |
| February | Longint | 2 | |
| Friday | Longint | 6 | |
| January | Longint | 1 | |
| July | Longint | 7 | |
| June | Longint | 6 | |
| March | Longint | 3 | |
| May | Longint | 5 | |
| Monday | Longint | 2 | |
| November | Longint | 11 | |
| October | Longint | 10 | |
| Saturday | Longint | 7 | |
| September | Longint | 9 | |
| Sunday | Longint | 1 | |
| Thursday | Longint | 5 | |
| Tuesday | Longint | 3 | |
| Wednesday | Longint | 4 | |

## Design Object Access

| Constant | Type | Value | Comment |
|---|---|---|---|
| Attribute executed on server | Longint | 8 | Corresponds to the "Execute on server" option |
| Attribute folder name | Longint | 1024 | Name of folder for the method ("folder" attribute). When you pass this constant, you must pass a folder name in *attribValue*:<br>• if this name corresponds to a valid folder, the method is placed in this parent folder,<br>• if the folder does not exist, the command does not change anything at the parent folder level,<br>• if you pass an empty string, the method is placed at the root level. |
| Attribute invisible | Longint | 1 | Corresponds to the "Invisible" option |
| Attribute published SOAP | Longint | 3 | Corresponds to the "Offered as a Web Service" option |
| Attribute published SQL | Longint | 7 | Corresponds to the "Available through SQL" option |
| Attribute published Web | Longint | 2 | Corresponds to the "Available through 4D HTML tags and URLs (4DACTION...)" option |
| Attribute published WSDL | Longint | 4 | Corresponds to the "Published in WSDL" option |
| Attribute shared | Longint | 5 | Corresponds to the "Shared by components and host database" option |
| Code with tokens | Longint | 1 | Include tokens in exported code |
| On object locked abort | Longint | 0 | Loading of object is aborted (default functioning) |
| On object locked confirm | Longint | 2 | 4D displays a dialog box so that you can choose to try again or to abort. In remote mode, this option is not supported (loading is aborted) |
| On object locked retry | Longint | 1 | 4D keeps attempting to load the object until it has been released |
| Path all objects | Longint | 31 | Paths of all the methods of the database |
| Path database method | Longint | 2 | Path of database methods specified (English name). List of these methods:<br>*[databaseMethod]/onStartup*<br>*[databaseMethod]/onExit*<br>*[databaseMethod]/onDrop*<br>*[databaseMethod]/onBackupStartup*<br>*[databaseMethod]/onBackupShutdown*<br>*[databaseMethod]/onWebConnection*<br>*[databaseMethod]/onWebAuthentication*<br>*[databaseMethod]/onWebSessionSuspend*<br>*[databaseMethod]/onServerStartup*<br>*[databaseMethod]/onServerShutdown*<br>*[databaseMethod]/onServerOpenConnexion*<br>*[databaseMethod]/onServerCloseConnection*<br>*[databaseMethod]/onSystemEvent*<br>*[databaseMethod]/onSqlAuthentication* |
| Path project form | Longint | 4 | Path of project form methods and all their object methods. Examples:<br>*[projectForm]/myForm/{formMethod}*<br>*[projectForm]/myForm/button1*<br>*[projectForm]/myForm/my%2list*<br>*[projectForm]/myForm/button1* |
| Path project method | Longint | 1 | Name of method.<br>Example: *MyProjectMethod* |
| Path table form | Longint | 16 | Path of table form methods and all their object methods. Example:<br>*[tableForm]/table_1/Form1/{formMethod}*<br>*[tableForm]/table_1/Form1/button1*<br>*[tableForm]/table_1/Form1/my%2list*<br>*[tableForm]/table_2/Form1/my%2list* |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Path trigger | Longint | 8 | Path of database triggers. Example:<br>*[trigger]/table_1*<br>*[trigger]/table_2* |

## Dictionaries

**Compatibility note:** These constants correspond to numbers for "Cordial" dictionaries, which are no longer supported in 4D starting with version 14. These constants are kept for reasons of compatibility (an equivalent Hunspell dictionary is used internally).

| Constant | Type | Value | Comment |
|---|---|---|---|
| English dictionary | Longint | 69632 | |
| French dictionary | Longint | 262144 | |
| German dictionary | Longint | 131584 | |
| Norwegian dictionary | Longint | 589824 | |
| Spanish dictionary | Longint | 196608 | |

## Digest Type

| Constant | Type | Value | Comment |
|---|---|---|---|
| 4D digest | Longint | 2 | Use internal algorithm of 4D |
| MD5 digest | Longint | 0 | Use the MD5 algorithm |
| SHA1 digest | Longint | 1 | Use the SHA-1algorithm |

## Euro Currencies

| Constant | Type | Value | Comment |
|---|---|---|---|
| Austrian Schilling | String | ATS | |
| Belgian Franc | String | BEF | |
| Deutsche Mark | String | DEM | |
| Euro | String | EUR | |
| Finnish Markka | String | FIM | |
| French Franc | String | FRF | |
| Greek Drachma | String | GRD | |
| Irish Pound | String | IEP | |
| Italian Lira | String | ITL | |
| Luxembourg Franc | String | LUF | |
| Netherlands Guilder | String | NLG | |
| Portuguese Escudo | String | PTE | |
| Spanish Peseta | String | ESP | |

## Events (Modifiers)

| Constant | Type | Value | Comment |
|---|---|---|---|
| Activate window bit | Longint | 0 | |
| Activate window mask | Longint | 1 | |
| Caps lock key bit | Longint | 10 | Windows and OS X |
| Caps lock key mask | Longint | 1024 | Windows and OS X |
| Command key bit | Longint | 8 | Ctrl key under Windows, Command key under OS X |
| Command key mask | Longint | 256 | Ctrl key under Windows, Command key under OS X |
| Control key bit | Longint | 12 | Ctrl key under OS X, or right click under Windows and OS X |
| Control key mask | Longint | 4096 | Ctrl key under OS X, or right click under Windows and OS X |
| Mouse button bit | Longint | 7 | |
| Mouse button mask | Longint | 128 | |
| Option key bit | Longint | 11 | Alt key (also called Option under OS X) |
| Option key mask | Longint | 2048 | Alt key (also called Option under OS X) |
| Right control key bit | Longint | 15 | |
| Right control key mask | Longint | 32768 | |
| Right option key bit | Longint | 14 | |
| Right option key mask | Longint | 16384 | |
| Right shift key bit | Longint | 13 | |
| Right shift key mask | Longint | 8192 | |
| Shift key bit | Longint | 9 | Windows and OS X |
| Shift key mask | Longint | 512 | Windows and OS X |

## Events (What)

| Constant | Type | Value | Comment |
|---|---|---|---|
| Activate event | Longint | 8 | |
| Auto key event | Longint | 5 | |
| Disk event | Longint | 7 | |
| Key down event | Longint | 3 | |
| Key up event | Longint | 4 | |
| Mouse down event | Longint | 1 | |
| Mouse up event | Longint | 2 | |
| Null event | Longint | 0 | |
| Operating system event | Longint | 15 | |
| Update event | Longint | 6 | |

## Expressions

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| MAXINT | Longint | 32767 | |
| MAXLONG | Longint | 2147483647 | |
| MAXTEXTLENBEFOREV11 | Longint | 32000 | |

## Field and Variable Types

| Constant | Type | Value | Comment |
|---|---|---|---|
| Array 2D | Longint | 13 | |
| Blob array | Longint | 31 | |
| Boolean array | Longint | 22 | |
| Date array | Longint | 17 | |
| Integer array | Longint | 15 | |
| Is alpha field | Longint | 0 | |
| Is BLOB | Longint | 30 | |
| Is Boolean | Longint | 6 | |
| Is date | Longint | 4 | |
| Is float | Longint | 35 | |
| Is integer | Longint | 8 | |
| Is integer 64 bits | Longint | 25 | |
| Is JSON null | Longint | 255 | |
| Is longint | Longint | 9 | |
| Is object | Longint | 38 | |
| Is picture | Longint | 3 | |
| Is pointer | Longint | 23 | |
| Is real | Longint | 1 | |
| Is string var | Longint | 24 | |
| Is subtable | Longint | 7 | |
| Is text | Longint | 2 | |
| Is time | Longint | 11 | |
| Is undefined | Longint | 5 | |
| LongInt array | Longint | 16 | |
| Object array | Longint | 39 | |
| Picture array | Longint | 19 | |
| Pointer array | Longint | 20 | |
| Real array | Longint | 14 | |
| String array | Longint | 21 | |
| Text array | Longint | 18 | |
| Time array | Longint | 32 | |

## Find Window

| Constant | Type | Value | Comment |
|---|---|---|---|
| In contents | Longint | 3 | Platform: Mac OS and Windows |
| In drag | Longint | 4 | Platform: Mac OS only (obsolete beginning with 4D v14) |
| In go away | Longint | 6 | Platform: Mac OS only (obsolete beginning with 4D v14) |
| In grow | Longint | 5 | Platform: Mac OS only (obsolete beginning with 4D v14) |
| In menu bar | Longint | 1 | Platform: Mac OS only (obsolete beginning with 4D v14) |
| In system window | Longint | 2 | Platform: Mac OS only (obsolete beginning with 4D v14) |
| In zoom box | Longint | 8 | Platform: Mac OS only (obsolete beginning with 4D v14) |

## Font Styles

Constants prefixed with _O_ are obsolete and must no longer be used.

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_Condensed | Longint | 32 | |
| _o_Extended | Longint | 64 | |
| _o_Outline | Longint | 8 | |
| _o_Shadow | Longint | 16 | |
| Automatic style sheet | String | __automatic__ | Used by default for all objects |
| Automatic style sheet_additional | String | __automatic_additional_text__ | Supported by static text, fields and variables only. Used for additional text in dialog boxes. |
| Automatic style sheet_main | String | __automatic_main_text__ | Supported by static text, fields and variables only. Used for main text in dialog boxes. |
| Bold | Longint | 1 | |
| Bold and Italic | Longint | 3 | |
| Bold and Underline | Longint | 5 | |
| Italic | Longint | 2 | |
| Italic and Underline | Longint | 6 | |
| Plain | Longint | 0 | |
| Underline | Longint | 4 | |

## Font Type List

| Constant | Type | Value | Comment |
|---|---|---|---|
| Favorite fonts | Longint | 1 | *fonts* contains the list of favorite fonts.<br>- Under Windows: list of active font family names.<br>- Under OS X: list of font family names found in the control panel, entitled "Favorites" in English, "Favoris" in French, "Favoriten" in German, and so on . This collection may be blank if the user has not added any favorite fonts. |
| Recent fonts | Longint | 2 | *fonts* contains the list of recent fonts (the ones used during the 4D session). This list is used in particular by multi-style text areas. |
| System fonts | Longint | 0 | *fonts* contains the list of all the system fonts. Default option when *listType* is omitted. |

## Form Area

| Constant | Type | Value | Comment |
|---|---|---|---|
| Form break0 | Longint | 300 | |
| Form break1 | Longint | 301 | |
| Form break2 | Longint | 302 | |
| Form break3 | Longint | 303 | |
| Form break4 | Longint | 304 | |
| Form break5 | Longint | 305 | |
| Form break6 | Longint | 306 | |
| Form break7 | Longint | 307 | |
| Form break8 | Longint | 308 | |
| Form break9 | Longint | 309 | |
| Form detail | Longint | 0 | |
| Form footer | Longint | 100 | |
| Form header | Longint | 200 | |
| Form header1 | Longint | 201 | |
| Form header10 | Longint | 210 | |
| Form header2 | Longint | 202 | |
| Form header3 | Longint | 203 | |
| Form header4 | Longint | 204 | |
| Form header5 | Longint | 205 | |
| Form header6 | Longint | 206 | |
| Form header7 | Longint | 207 | |
| Form header8 | Longint | 208 | |
| Form header9 | Longint | 209 | |

## Form Events

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_On Mac toolbar button | Longint | 55 | *** Obsolete constant *** |
| On Activate | Longint | 11 | The form’s window becomes the frontmost window |
| On After Edit | Longint | 45 | The contents of the enterable object that has the focus has just been modified |
| On After Keystroke | Longint | 28 | A character is about to be entered in the object that has the focus. **Get edited text** returns the object's text including this character. |
| On After Sort | Longint | 30 | (*List box only*) A standard sort has just been carried out in a list box column |
| On Alternative Click | Longint | 38 | • *3D buttons*: The “arrow” area of a 3D button is clicked<br>• *List boxes*: In a column of an object array, an ellipsis button ("alternateButton" attribute) is clicked<br>**Note**: Ellipsis buttons are only available for versions v15 or higher. |
| On Before Data Entry | Longint | 41 | (*List box only*) A list box cell is about to change to editing mode |
| On Before Keystroke | Longint | 17 | A character is about to be entered in the object that has the focus. **Get edited text** returns the object's text without this character. |
| On Begin Drag Over | Longint | 46 | An object is being dragged |
| On Begin URL Loading | Longint | 47 | (*Web areas only*) A new URL is loaded in the Web area |
| On bound variable change | Longint | 54 | The variable bound to a subform is modified. |
| On Clicked | Longint | 4 | A click occurred on an object |
| On Close Box | Longint | 22 | The window’s close box has been clicked |
| On Close Detail | Longint | 26 | You left the detail form and are going back to the output form |
| On Collapse | Longint | 44 | (*Hierarchical lists and hierarchical list boxes*) An element of the hierarchical list or hierarchical list box has been collapsed using a click or a keystroke |
| On Column Moved | Longint | 32 | (*List box only*) A list box column is moved by the user via drag and drop |
| On Column Resize | Longint | 33 | (*List box only*) The width of a list box column is modified by a user with the mouse |
| On Data Change | Longint | 20 | Object data has been modified |
| On Deactivate | Longint | 12 | The form’s window ceases to be the frontmost window |
| On Delete Action | Longint | 58 | (*Hierarchical lists and List boxes*) The user attempts to delete an item |
| On Display Detail | Longint | 8 | A record is about to be displayed in a list or a row is about to be displayed in a list box. |
| On Double Clicked | Longint | 13 | A double click occurred on an object |
| On Drag Over | Longint | 21 | Data could be dropped onto an object |
| On Drop | Longint | 16 | Data has been dropped onto an object |
| On End URL Loading | Longint | 49 | (*Web areas only*) All the resources of the URL have been loaded |
| On Expand | Longint | 43 | (*Hierarchical lists and hierarchical list boxes*) An element of the hierarchical list or hierarchical list box has been expanded using a click or a keystroke |

| Constant | Type | Value | Comment |
|---|---|---|---|
| On Footer Click | Longint | 57 | (*List boxes only*) A click occurs in the footer of a list box or a list box column |
| On Getting Focus | Longint | 15 | A form object is getting the focus |
| On Header | Longint | 5 | The form's header area is about to be printed or displayed |
| On Header Click | Longint | 42 | (*List box only*) A click occurs in a column header of the list box |
| On Load | Longint | 1 | The form is about to be displayed or printed |
| On Load Record | Longint | 40 | During entry in list, a record is loaded during modification (the user clicks on a record line and a field changes to editing mode) |
| On Long Click | Longint | 39 | (*3D buttons only*) A 3D button is clicked and the mouse button remains pushed for a certain lapse of time |
| On Losing Focus | Longint | 14 | A form object is losing the focus |
| On Menu Selected | Longint | 18 | A menu item has been chosen |
| On Mouse Enter | Longint | 35 | The mouse cursor enters the graphic area of an object |
| On Mouse Leave | Longint | 36 | The mouse cursor leaves the graphic area of an object |
| On Mouse Move | Longint | 37 | The mouse cursor moves at least one pixel OR a modifier key (Shift, Alt, Shift Lock) was pressed. If the event is checked for an object only, it is generated only when the cursor is within the graphic area of the object |
| On Mouse Up | Longint | 2 | (*Pictures only*) The user has just released the left mouse button in a Picture object |
| On Open Detail | Longint | 25 | The detail form associated with the output form or with the listbox is about to be opened |
| On Open External Link | Longint | 52 | (*Web areas only*) An external URL has been opened in the browser |
| On Outside Call | Longint | 10 | The form received a **CALL PROCESS** call |
| On Page Change | Longint | 56 | The current page of the form is changed |
| On Plug in Area | Longint | 19 | An external object requested its object method to be executed |
| On Printing Break | Longint | 6 | One of the form's break areas is about to be printed |
| On Printing Detail | Longint | 23 | The form's detail area is about to be printed |
| On Printing Footer | Longint | 7 | The form's footer area is about to be printed |
| On Resize | Longint | 29 | The form window is resized |
| On Row Moved | Longint | 34 | (*List box only*) A list box row is moved by the user via drag and drop |
| On Scroll | Longint | 59 | The user scrolls the contents of a picture field or variable using the mouse or keyboard. |
| On Selection Change | Longint | 31 | • *List box*: The current selection of rows or columns is modified<br>• *Records in list*: The current record or the current selection of rows is modified in a list form or subform<br>• *Hierarchical list*: The selection in the list is modified following a click or a keystroke<br>• *Enterable field or variable*: The text selection or the position of the cursor in the area is modified following a click or a keystroke |
| On Timer | Longint | 27 | The number of ticks defined by the **SET TIMER** command has passed |
| On Unload | Longint | 24 | The form is about to be exited and released |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| On URL Filtering | Longint | 51 | (*Web areas only*) A URL was blocked by the Web area |
| On URL Loading Error | Longint | 50 | (*Web areas only*) An error occurred when the URL was loading |
| On URL Resource Loading | Longint | 48 | (*Web areas only*) A new resource is loaded in the Web area |
| On Validate | Longint | 3 | The record data entry has been validated |
| On Window Opening Denied | Longint | 53 | (*Web areas only*) A pop-up window has been blocked |

## Form Object Types

| Constant | Type | Value | Comment |
|---|---|---|---|
| Object type 3D button | Longint | 16 | |
| Object type 3D checkbox | Longint | 26 | |
| Object type 3D radio button | Longint | 23 | |
| Object type button grid | Longint | 20 | |
| Object type checkbox | Longint | 25 | |
| Object type combobox | Longint | 11 | |
| Object type dial | Longint | 28 | |
| Object type group | Longint | 21 | |
| Object type groupbox | Longint | 30 | |
| Object type hierarchical list | Longint | 6 | |
| Object type hierarchical popup menu | Longint | 13 | |
| Object type highlight button | Longint | 17 | |
| Object type invisible button | Longint | 18 | |
| Object type line | Longint | 32 | |
| Object type listbox | Longint | 7 | |
| Object type listbox column | Longint | 9 | |
| Object type listbox footer | Longint | 10 | |
| Object type listbox header | Longint | 8 | |
| Object type matrix | Longint | 35 | |
| Object type oval | Longint | 34 | |
| Object type picture button | Longint | 19 | |
| Object type picture input | Longint | 4 | |
| Object type picture popup menu | Longint | 14 | |
| Object type picture radio button | Longint | 24 | |
| Object type plugin area | Longint | 38 | |
| Object type popup dropdown list | Longint | 12 | |
| Object type progress indicator | Longint | 27 | |
| Object type push button | Longint | 15 | |
| Object type radio button | Longint | 22 | |
| Object type radio button field | Longint | 5 | |
| Object type rectangle | Longint | 31 | |
| Object type rounded rectangle | Longint | 33 | |
| Object type ruler | Longint | 29 | |
| Object type splitter | Longint | 36 | |
| Object type static picture | Longint | 2 | |
| Object type static text | Longint | 1 | |
| Object type subform | Longint | 39 | |
| Object type tab control | Longint | 37 | |
| Object type text input | Longint | 3 | |
| Object type unknown | Longint | 0 | |
| Object type web area | Longint | 40 | |
| Object type write pro area | Longint | 41 | |

## Form Objects (Access)

| Constant | Type | Value | Comment |
|---|---|---|---|
| Form all pages | Longint | 2 | Returns all objects of all the pages, excluding inherited objects |
| Form current page | Longint | 1 | Returns all objects of the current page, including page 0 but excluding inherited objects |
| Form inherited | Longint | 4 | Returns inherited objects only |
| Object current | Longint | 0 | |
| Object first in entry order | String | ;FirstObject | |
| Object named | Longint | 3 | |
| Object subform container | Longint | 2 | |
| Object with focus | Longint | 1 | |

## Form Objects (Properties)

| Constant | Type | Value | Comment |
|---|---|---|---|
| Align bottom | Longint | 4 | |
| Align center | Longint | 3 | |
| Align default | Longint | 1 | |
| Align left | Longint | 2 | |
| Align right | Longint | 4 | |
| Align top | Longint | 2 | |
| Asynchronous progress bar | Longint | 3 | Circular indicator displaying continuous animation |
| Barber shop | Longint | 2 | Bar displaying continuous animation |
| Border Dotted | Longint | 2 | Objects appear framed with a dotted 1-pt. border line |
| Border Double | Longint | 5 | Objects appear framed with a double line, i.e., two continuous 1-pt. lines separated by a pixel |
| Border None | Longint | 0 | Objects appear with no border |
| Border Plain | Longint | 1 | Objects appear framed with a continuous 1-pt. border line |
| Border Raised | Longint | 3 | Objects appear framed with a 3D effect (raised) |
| Border Sunken | Longint | 4 | Objects appear framed with a sunken 3D effect |
| Border System | Longint | 6 | The border line is drawn based on the graphic specifications of the system |
| Choice list | Longint | 0 | Simple list of values to choose from ("Choice List" option in the Property List) (default) |
| Disable events others unchanged | Longint | 2 | All the events listed in the *arrEvents* array are disabled; the status of other events remain unchanged |
| Enable events disable others | Longint | 0 | All the events listed in the *arrEvents* array are enabled; all the other events are disabled |
| Enable events others unchanged | Longint | 1 | All the events listed in the *arrEvents* array are enabled; the status of other events remain unchanged |
| Excluded list | Longint | 2 | Lists values not accepted for entry ("Excluded List" option in the Property List) |
| Multiline Auto | Longint | 0 | In single-line areas, words located at the end of lines are truncated and there are no line returns.<br>In multiline areas, 4D carries out automatic line returns. |
| Multiline No | Longint | 2 | There are never line returns: the text is always displayed on a single row. If the Alpha or Text field or variable contains carriage returns, the text located after the first carriage return is removed as soon as the area is modified. |
| Multiline Yes | Longint | 1 | In single-line areas, the text is displayed up to the first carriage return or until the last word that can be displayed entirely. 4D inserts line returns; it is possible to scroll the contents of the area by pressing the down arrow key.<br>In multiline areas, 4D carries out automatic line returns. |
| Orientation 0° | Longint | 0 | No rotation (default value) |
| Orientation 180° | Longint | 180 | Orientation of text to 180° clockwise |
| Orientation 90° left | Longint | 270 | Orientation of text to 90° counter-clockwise |
| Orientation 90° right | Longint | 90 | Orientation of text to 90° clockwise |
| Print Frame fixed with multiple records | Longint | 2 | The frame remains the same size, but 4D prints the form several times to include all the records. |
| Print Frame fixed with truncation | Longint | 1 | 4D prints only the records that fit into the area of the subform. The form is printed only once and those records that are not printed are ignored. |
| Progress bar | Longint | 1 | Standard progress bar |
| Required list | Longint | 1 | Lists only values accepted for entry ("Required List" option in the Property List) |
| Resize horizontal grow | Longint | 1 | If the window grows by 50% in width, the object is expanded by 50% to the right. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Resize horizontal move | Longint | 2 | If the window grows by 100 pixels in width, the object is moved 100 pixels to the right. |
| Resize horizontal none | Longint | 0 | If the window is expanded in width, neither the width nor the position of the object changes. |
| Resize vertical grow | Longint | 1 | If the window grows by 50% in height, the object is lengthened by 50% towards the bottom. |
| Resize vertical move | Longint | 2 | If the window grows by 100 pixels in height, the object is moved 100 pixels towards the bottom. |
| Resize vertical none | Longint | 0 | If the window is expanded in height, neither the height nor the position of the object changes. |

## Form Parameters

| Constant | Type | Value | Comment |
|---|---|---|---|
| Multiple selection | Longint | 2 | The user can select several records at once. To select adjacent records, click on the first record to be selected, then press the **Shift** key before clicking on the last record you want to include in the selection. To select non-adjacent records, click on each record separately while holding down the **Ctrl** (under Windows) or **Command** (under Mac OS) key. |
| No selection | Longint | 0 | It is not be possible to select a record in the list |
| NonInverted objects | Longint | 0 | |
| Single selection | Longint | 1 | Only one record can be selected at a time |

## Function Keys

| Constant | Type | Value | Comment |
|---|---|---|---|
| Backspace key | Longint | 8 | |
| Down arrow key | Longint | 31 | |
| End key | Longint | 4 | |
| Enter key | Longint | 3 | |
| Escape key | Longint | 27 | |
| F1 key | Longint | -122 | |
| F10 key | Longint | -109 | |
| F11 key | Longint | -103 | |
| F12 key | Longint | -111 | |
| F13 key | Longint | -105 | |
| F14 key | Longint | -107 | |
| F15 key | Longint | -113 | |
| F2 key | Longint | -120 | |
| F3 key | Longint | -99 | |
| F4 key | Longint | -118 | |
| F5 key | Longint | -96 | |
| F6 key | Longint | -97 | |
| F7 key | Longint | -98 | |
| F8 key | Longint | -100 | |
| F9 key | Longint | -101 | |
| Help key | Longint | 5 | |
| Home key | Longint | 1 | |
| Left arrow key | Longint | 28 | |
| Page down key | Longint | 12 | |
| Page up key | Longint | 11 | |
| Return key | Longint | 13 | |
| Right arrow key | Longint | 29 | |
| Tab key | Longint | 9 | |
| Up arrow key | Longint | 30 | |

## Graph Parameters

| Constant | Type | Value | Comment |
|---|---|---|---|
| Graph background color | String | graphBackgroundColor | **Possible values:** SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414" |
| Graph background opacity | String | graphBackgroundOpacity | **Possible values**: Integers, range 0-100<br>**Default value**: 100 |
| Graph background shadow color | String | graphBackgroundShadowColor | **Possible values**: SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414" |
| Graph bottom margin | String | bottomMargin | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph colors | String | colors | **Possible values**: Text array. Colors for each graph series.<br>**Default values**: Blue-green (#19BAC9), Yellow (#FFC338), Purple (#573E82), Green (#4FA839), Orange (#D95700), Blue (#1D9DF2), Yellow-green (#B5CF32), Red (#D43A26) |
| Graph column gap | String | columnGap | **Possible value**s: Longints<br>**Default value**: 12<br>Sets spacing between bars |
| Graph column width max | String | columnWidthMax | **Possible values**: Real numbers<br>**Default value**: 200 |
| Graph column width min | String | columnWidthMin | **Possible values**: Real numbers<br>**Default value**: 10 |
| Graph default height | String | defaultHeight | **Possible values**: Real numbers<br>**Default value**: 400. If graphType=7 (Pie), then default value = 600 |
| Graph default width | String | defaultWidth | **Possible values**: Real numbers<br>**Default value**: 600. If graphType=7 (Pie), then default value = 800 |
| Graph display legend | String | displayLegend | **Possible value**s: Boolean<br>**Default value**: True |
| Graph document background color | String | documentBackgroundColor | **Possible values**: SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414". When a graph saved as an SVG picture is opened elsewhere, the document background color is only displayed if the SVG rendering engine supports the *SVG tiny 1.2* norm (supported on IE, Firefox, but not on Chrome). |
| Graph document background opacity | String | documentBackgroundOpacity | **Possible values**: Integer, range 0-100 (default value: 100). When a graph saved as an SVG picture is opened elsewhere, the document background opacity is only displayed if the SVG rendering engine supports the *SVG tiny 1.2* norm (supported on IE, Firefox, but not on Chrome). |
| Graph font color | String | fontColor | **Possible values:** SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414" |
| Graph font size | String | fontSize | **Possible values**: Longints<br>**Default value**: 12. If graphType=7 (Pie), see Graph pie font size |
| Graph left margin | String | leftMargin | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph legend font color | String | legendFontColor | **Possible values**: SVG compliant color expression (text), for example "#7F8E00", "Pink", or "#0a1414" |
| Graph legend icon gap | String | legendIconGap | **Possible values**: Real numbers<br>**Default value**: Graph legend icon height/2 |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Graph legend icon height | String | legendIconHeight | **Possible values**: Real numbers<br>**Default value**: 20 |
| Graph legend icon width | String | legendIconWidth | **Possible values**: Real numbers<br>**Default value**: 20 |
| Graph legend labels | String | legendLabels | **Possible values**: Text array. If missing, 4D displays icons without text. |
| Graph line width | String | lineWidth | **Possible values**: Real numbers<br>**Default value**: 2 |
| Graph pie font size | String | pieFontSize | **Possible values**: Real numbers<br>**Default value**: 16 |
| Graph pie shift | String | pieShift | **Possible values**: Real numbers<br>**Default value**: 8 |
| Graph plot height | String | plotHeight | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph plot radius | String | plotRadius | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph plot width | String | plotWidth | **Possible values**: Real numbers<br>**Default value**: 12 |
| Graph right margin | String | rightMargin | **Possible values**: Real numbers<br>**Default value**: 2 |
| Graph top margin | String | topMargin | **Possible values**: Real numbers<br>**Default value**: 2 |
| Graph type | String | graphType | **Possible values**: Longints [1 to 8] where 1 = bars, 2 = proportional, 3 = stacked, 4 = lines, 5 = surfaces, 6 = scatter, 7 = pie, 8 = pictures.<br>**Default value**: 1<br>If null, the graph is not drawn and no error message is displayed. If out of range, the graph is not drawn and an error message is displayed.<br>If you want to modify picture type graphs (value=8), you must copy the 4D/Resources/GraphTemplates/Graph_8_Pictures/ folder into the Resources folder of your database and perform the necessary modifications. Local picture files will be used instead of 4D files. There is no pattern for picture names; 4D sorts the files contained in the folder and assigns the first file to the first graph. These files can be of the SVG or image type. |
| Graph xGrid | String | xGrid | **Possible values**: Boolean<br>**Default value**: True.<br>Used only with proportional types 4 and 6. |
| Graph xMax | String | xMax | **Possible values**: Number, Date, Time (same type as *xLabels* parameter).<br>Only values lower than xMax are displayed on the graph. xMax is used only for graph types 4, 5, or 6 if xProp=true and if *xLabels* type is a number, date, or time. If missing or if xMin>xMax, 4D automatically calculates the xMax value. |
| Graph xMin | String | xMin | **Possible values**: Number, Date, Time (same type as *xLabels* parameter).<br>Only values higher than xMin are displayed on the graph. xMin is used only for graph types 4, 5, or 6 if xProp=true and if *xLabels* type is a number, date, or time. If missing or if xMin>xMax, 4D automatically calculates the xMin value. |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Graph xProp | String | xProp | **Possible values**: Boolean<br>**Default value**: False<br>True for proportional x-axis; False for normal x-axis. xProp is used only for graph types 4, 5, or 6 |
| Graph yGrid | String | yGrid | **Possible values**: Boolean<br>**Default value**: True |
| Graph yMax | String | yMax | **Possible value**s: Numbers<br>If missing, 4D automatically calculates the yMax value. |
| Graph yMin | String | yMin | **Possible values**: Numbers<br>If missing, 4D automatically calculates the yMin value. |

## Hierarchical Lists

| Constant | Type | Value | Comment |
|---|---|---|---|
| Additional text | String | 4D_additional_text | |
| Ala Macintosh | Longint | 1 | |
| Ala Windows | Longint | 2 | |
| Macintosh node | Longint | 860 | |
| Use PicRef | Longint | 131072 | |
| Use PICT resource | Longint | 65536 | |
| Windows node | Longint | 138 | |

## HTTP Client

| Constant | Type | Value | Comment |
|---|---|---|---|
| HTTP basic | Longint | 1 | Use BASIC authentication method |
| HTTP compression | Longint | 6 | *value* = 0 (do not compress) or 1 (compress). Default value: 0<br>This option enables or disables the compression mechanism intended to accelerate exchanges for requests between the client and server. When this mechanism is enabled, the HTTP client uses deflate or gzip compression depending on the server response. |
| HTTP DELETE method | String | DELETE | See RFC 2616 |
| HTTP digest | Longint | 2 | Use DIGEST authentication method |
| HTTP display auth dial | Longint | 4 | *value* = 0 (do not display dialogue box) or 1 (display dialogue box). Default value: 0<br>This option displays the authentication dialog box when the **HTTP Get** or **HTTP Request** command is executed. By default, this command never displays the dialog box and you must normally use the **HTTP AUTHENTICATE** command. However, if you want an authentication dialog box to appear so that users can enter their identifiers, then pass 1 in value. The dialog box only appears when the request requires authentication. |
| HTTP follow redirect | Longint | 2 | *value* = 0 (do not accept redirections) or 1 (accept redirections).<br>Default value = 1 |
| HTTP GET method | String | GET | See RFC 2616. Same as using **HTTP Get** command. |
| HTTP HEAD method | String | HEAD | See RFC 2616 |
| HTTP max redirect | Longint | 3 | *value* = Maximum number of redirections accepted<br>Default value = 2 |
| HTTP OPTIONS method | String | OPTIONS | See RFC 2616 |
| HTTP POST method | String | POST | See RFC 2616 |
| HTTP PUT method | String | PUT | See RFC 2616 |
| HTTP reset auth settings | Longint | 5 | *value* = 0 (do not delete information) or 1 (delete information). Default value: 0<br>This option indicates to 4D to reset the authentication information of the user (user name, password, method) after each execution of the **HTTP Get** or **HTTP Request** command in the same process. By default, this information is kept and reused for each request. Pass 1 in *value* to delete this information after each call. Note that regardless of the setting, this information is deleted when the process is killed. |
| HTTP timeout | Longint | 1 | *value* = timeout of client request, expressed in seconds. This timeout sets how long the HTTP client waits for the server to respond. After this period of time has passed, the client closes the session and the request is lost.<br>By default, this timeout is 120 seconds. It can be changed because of specific characteristics (network state, request characteristics, etc.). |
| HTTP TRACE method | String | TRACE | See RFC 2616 |

## Index Type

| Constant | Type | Value | Comment |
|---|---|---|---|
| Cluster BTree index | Longint | 3 | B-Tree type index using clusters. This type of index is optimized when the index contains few keywords, i.e. when the same values occur frequently in the data. |
| Default index type | Longint | 0 | 4D specifies the index type (excluding keywords indexes) that is the most optimized according to the contents of the field. |
| Keywords index | Longint | -1 | Permits word-by-word indexing of field contents. This type of index can only be used with fields of the Text, Alpha or Picture type. Warning: Keywords indexes cannot be composite. |
| Standard BTree index | Longint | 1 | Standard B-Tree type index. This multi-purpose index type is used in previous versions of 4D |

## Is License Available

| Constant | Type | Value | Comment |
|---|---|---|---|
| 4D Client SOAP license | Longint | 808465465 | |
| 4D Client Web license | Longint | 808465209 | |
| 4D Draw license | Longint | 808464694 | |
| 4D for ADO license | Longint | 808465714 | |
| 4D for MySQL license | Longint | 808465712 | |
| 4D for OCI license | Longint | 808465208 | |
| 4D for PostgreSQL license | Longint | 808465713 | |
| 4D for Sybase license | Longint | 808465715 | |
| 4D Mobile license | Longint | 808464439 | |
| 4D Mobile Test license | Longint | 808465719 | |
| 4D ODBC Pro license | Longint | 808464946 | |
| 4D SOAP license | Longint | 808465464 | |
| 4D SOAP local license | Longint | 808531000 | |
| 4D SOAP one connection license | Longint | 825242680 | |
| 4D SQL Server license | Longint | 808464949 | |
| 4D SQL Server local license | Longint | 808530485 | |
| 4D SQL Server one conn. license | Longint | 825242165 | |
| 4D View license | Longint | 808465207 | |
| 4D Web license | Longint | 808464945 | |
| 4D Web local license | Longint | 808530481 | |
| 4D Web one connection license | Longint | 825242161 | |
| 4D Write license | Longint | 808464697 | |

## ISO Latin Character Entities

| Constant | Type | Value | Comment |
|---|---|---|---|
| ISO L1 a acute | String | &aacute; | |
| ISO L1 a circumflex | String | &acirc; | |
| ISO L1 a grave | String | &agrave; | |
| ISO L1 a ring | String | &aring; | |
| ISO L1 a tilde | String | &atilde; | |
| ISO L1 a umlaut | String | &auml; | |
| ISO L1 ae ligature | String | &aelig; | |
| ISO L1 Ampersand | String | &amp; | |
| ISO L1 c cedilla | String | &ccedil; | |
| ISO L1 Cap A acute | String | &Aacute; | |
| ISO L1 Cap A circumflex | String | &Acirc; | |
| ISO L1 Cap A grave | String | &Agrave; | |
| ISO L1 Cap A ring | String | &Aring; | |
| ISO L1 Cap A tilde | String | &Atilde; | |
| ISO L1 Cap A umlaut | String | &Auml; | |
| ISO L1 Cap AE ligature | String | &AELig; | |
| ISO L1 Cap C cedilla | String | &Ccedil; | |
| ISO L1 Cap E acute | String | &Eacute; | |
| ISO L1 Cap E circumflex | String | &Ecirc; | |
| ISO L1 Cap E grave | String | &Egrave; | |
| ISO L1 Cap E umlaut | String | &Euml; | |
| ISO L1 Cap Eth Icelandic | String | &ETH; | |
| ISO L1 Cap I acute | String | &Iacute; | |
| ISO L1 Cap I circumflex | String | &Icirc; | |
| ISO L1 Cap I grave | String | &Igrave; | |
| ISO L1 Cap I umlaut | String | &Iuml; | |
| ISO L1 Cap N tilde | String | &Ntilde; | |
| ISO L1 Cap O acute | String | &Oacute; | |
| ISO L1 Cap O circumflex | String | &Ocirc; | |
| ISO L1 Cap O grave | String | &Ograve; | |
| ISO L1 Cap O slash | String | &Oslash; | |
| ISO L1 Cap O tilde | String | &Otilde; | |
| ISO L1 Cap O umlaut | String | &Ouml; | |
| ISO L1 Cap THORN Icelandic | String | &THORN; | |
| ISO L1 Cap U acute | String | &Uacute; | |
| ISO L1 Cap U circumflex | String | &Ucirc; | |
| ISO L1 Cap U grave | String | &Ugrave; | |
| ISO L1 Cap U umlaut | String | &Uuml; | |
| ISO L1 Cap Y acute | String | &Yacute; | |
| ISO L1 Copyright | String | &copy; | |
| ISO L1 e acute | String | &eacute; | |
| ISO L1 e circumflex | String | &ecirc; | |
| ISO L1 e grave | String | &egrave; | |
| ISO L1 e umlaut | String | &euml; | |
| ISO L1 eth Icelandic | String | &eth; | |
| ISO L1 Greater than | String | &gt; | |
| ISO L1 i acute | String | &iacute; | |
| ISO L1 i circumflex | String | &icirc; | |
| ISO L1 i grave | String | &igrave; | |
| ISO L1 i umlaut | String | &iuml; | |
| ISO L1 Less than | String | &lt; | |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| ISO L1 n tilde | String | ñ | |
| ISO L1 o acute | String | ó | |
| ISO L1 o circumflex | String | ô | |
| ISO L1 o grave | String | ò | |
| ISO L1 o slash | String | ø | |
| ISO L1 o tilde | String | õ | |
| ISO L1 o umlaut | String | ö | |
| ISO L1 Quotation mark | String | " | |
| ISO L1 Registered | String | ® | |
| ISO L1 sharp s German | String | ß | |
| ISO L1 thorn Icelandic | String | þ | |
| ISO L1 u acute | String | ú | |
| ISO L1 u circumflex | String | û | |
| ISO L1 u grave | String | ù | |
| ISO L1 u umlaut | String | ü | |
| ISO L1 y acute | String | ý | |
| ISO L1 y umlaut | String | ÿ | |

## LDAP

| Constant | Type | Value | Comment |
|---|---|---|---|
| LDAP all levels | String | sub | Search in the root entry level defined by *dnRootEntry* and in all subsequent entries |
| LDAP password MD5 | Longint | 0 | (Default) Send password encrypted in MD5 |
| LDAP password plain text | Longint | 1 | Send password with no encryption (TLS connection recommended) |
| LDAP root and next | String | one | Search in the root entry level defined by *dnRootEntry* and in the directly subsequent entries on one level |
| LDAP root only | String | base | Search only in the root entry level defined by *dnRootEntry* (default if omitted) |

## List Box

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| lk add to selection | Longint | 1 | The row selected is added to the existing selection. If the row specified already belongs to the existing selection, the command does nothing. |
| lk all | Longint | 0 | The command affects all sub-levels (default value, used when parameter is omitted). |
| lk background color | Longint | 1 | |
| lk background color array | Longint | 1 | |
| lk break row | Longint | 2 | The command affects the sub-level to which the "cell" designated by the *row* and *column* parameters belongs. Note that these parameters represent the row and column numbers in the list box in standard mode and not in its hierarchical representation. If the *row* and *column* parameters are omitted, the command does nothing. |
| lk control array | Longint | 3 | |
| lk display footer | Longint | 8 | **Display Footers** property<br>Applies to: List box<br>Possible values:<br>• lk no (0): hidden<br>• lk yes (1): shown |
| lk display header | Longint | 0 | **Display Headers** property<br>Applies to: List box<br>Possible values:<br>• lk no (0): hidden<br>• lk yes (1): shown |
| lk display hor scrollbar | Longint | 2 | 0=hidden, 1=shown |
| lk display ver scrollbar | Longint | 4 | 0=hidden, 1=shown |
| lk font color | Longint | 0 | |
| lk font color array | Longint | 0 | |
| lk footer height | Longint | 9 | Height in pixels |
| lk header height | Longint | 1 | Height in pixels |
| lk hor scrollbar height | Longint | 3 | Height in pixels |
| lk hor scrollbar position | Longint | 6 | Position of the cursor in pixels |
| lk inherited | Longint | -255 | |
| lk last printed row number | Longint | 0 | Returns in *info* the number of the last row printed. Lets you find out the number of the next row to be printed.<br>The number returned may be greater than the number of rows actually printed if the list box contains invisible rows or if the **OBJECT SET SCROLL POSITION** command has been called. For example, if rows 1, 18 and 20 have been printed, *info* is 20. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| lk level | Longint | 3 | The command affects all the break rows corresponding to the *level* column. This parameter designates a column number in the list box in standard mode and not in its hierarchical representation. If the *level* parameter is omitted, the command does nothing. |
| lk lines | Longint | 1 | Height designates a number of lines. 4D calculates the height of a line according to the font |
| lk pixels | Longint | 0 | Height is a number of pixels (default). |
| lk printed height | Longint | 3 | Returns in *info* the height in pixels of the object actually printed (including headers, lines, etc.). Remember that if the number of rows to print is less than the "capacity" of the list box, its height is automatically reduced. |
| lk printed rows | Longint | 1 | Returns in *info* the number of rows actually printed during the last call to the **Print object** command. This number includes any break rows added in the case of a hierarchical list box. For example, *info* is 10 if the list box contains 20 rows and the odd-numbered rows were hidden. |
| lk printing is over | Longint | 2 | Returns in *info* a Boolean indicating whether the last (visible) row of the list box has actually been printed. True = row has been printed; Otherwise, False. |
| lk remove from selection | Longint | 2 | The row selected is removed from the existing selection. If the row specified does not belong to the existing selection, the command does nothing. |
| lk replace selection | Longint | 0 | The row selected becomes the new selection and replaces the existing selection. The command has the same effect as a user click on a row (however, the On Clicked event is not generated). This is the default action (if the *action* parameter is omitted). |
| lk row height array | Longint | 4 | (4D View Pro license required) |
| lk row is disabled | Longint | 2 | The corresponding row is disabled. The text and controls such as check boxes are dimmed or grayed out. Enterable text input areas are no longer enterable. Default value: Enabled |
| lk row is hidden | Longint | 1 | The corresponding row is hidden. Hiding rows only affects the display of the list box. The hidden rows are still present in the arrays and can be managed by programming. The language commands, more particularly **LISTBOX Get number of rows** or **LISTBOX GET CELL POSITION**, do not take the displayed/hidden status of rows into account. For example, in a list box with 10 rows where the first 9 rows are hidden, **LISTBOX Get number of rows** returns 10. From the user's point of view, the presence of hidden rows in a list box is not visibly discernible. Only visible rows can be selected (for example using the Select All command). Default value: Visible |
| lk row is not selectable | Longint | 4 | The corresponding row is not selectable (highlighting is not possible). Enterable text input areas are no longer enterable unless the "Single-Click Edit" option is enabled. Controls such as check boxes and lists are still functional however. This setting is ignored if the list box selection mode is "None". Default value: Selectable |
| lk selection | Longint | 1 | The command affects selected sub-levels. |
| lk style array | Longint | 2 | |
| lk ver scrollbar position | Longint | 7 | Position of the cursor in pixels |
| lk ver scrollbar width | Longint | 5 | Width in pixels |

## Listbox Footer Calculation

| Constant | Type | Value | Comment |
|---|---|---|---|
| Listbox footer std deviation | Longint | 7 | Used with number or time type columns (only for array type list boxes)<br>Default type of the result: Real |
| lk footer average | Longint | 6 | Used with number or time type columns<br>Default type of the result: Real |
| lk footer count | Longint | 5 | Used with number, text, date, time, Boolean or picture type columns<br>Default type of the result: Longint |
| lk footer custom | Longint | 1 | No calculation performed by 4D. The footer variable must be calculated by programming.<br>Default type of the result: Footer variable type |
| lk footer max | Longint | 3 | Used with number, date, time or Boolean type columns<br>Default type of the result: Column array or field type |
| lk footer min | Longint | 2 | Used with number, date, time or Boolean type columns<br>Default type of the result: Column array or field type |
| lk footer sum | Longint | 4 | Used with number, time or Boolean type columns<br>Default type of the result: Column array or field type |
| lk footer sum squares | Longint | 9 | Used with number or time type columns (only for array type list boxes)<br>Default type of the result: Real |
| lk footer variance | Longint | 8 | Used with number or time type columns (only for array type list boxes)<br>Default type of the result: Real |

## Log Events

| Constant | Type | Value | Comment |
|---|---|---|---|
| Error message | Longint | 2 | |
| Information message | Longint | 0 | |
| Into 4D commands log | Longint | 3 | Indicates to 4D to record the *message* in the 4D commands log file, if this file has been activated. The 4D commands log file can be enabled using the **SET DATABASE PARAMETER** command (selector 34).<br>**Note:** 4D log files are grouped together in the **Logs** folder, which is created next to the database structure file (see the **Get 4D folder** command). |
| Into 4D debug message | Longint | 1 | Indicates to 4D to send the *message* to the system debugging environment. The result depends on the platform:<br>• Under Mac OS: the command sends the message to the Console<br>• Under Windows: the command sends the message as a debug message. To be able to read this message, you must have Microsoft Visual Studio or the DebugView utility for Windows (http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx) |
| Into 4D diagnostic log | Longint | 5 | Indicates to 4D to record the *message* in the 4D diagnostic file, if this file has been enabled. The diagnostic log file can be enabled using the **SET DATABASE PARAMETER** command (selector 79). |
| Into 4D request log | Longint | 2 | Indicates to 4D to record the *message* in the 4D requests log, if this file has been activated |
| Into Windows log events | Longint | 0 | Indicates to 4D to send the *message* to the “Log events” of Windows. This log receives and stores messages coming from running applications. In this case, you can attribute a level of importance to *message* via the optional *importance* parameter (see below).<br>**Notes:**<br>• For this feature to be available, the Windows Log Events service must be running.<br>• Under Mac OS, the command does nothing with this output type |
| Warning message | Longint | 1 | |

## Math

| Constant | Type | Value | Comment |
|---|---|---|---|
| Degree | Real | 0.0174532925199432958 | |
| e number | Real | 2.71828182845904524 | |
| Pi | Real | 3.141592653589793239 | |
| Radian | Real | 57.29577951308232088 | |

## Menu Item Properties

| Constant | Type | Value | Comment |
|---|---|---|---|
| Access privileges | String | 4D_access_group | Assign an access group to the command<br>0 = All Groups<br>>0 = Group ID |
| Associated standard action | String | 4D_standard_action | Associate a standard action with a menu item<br>See the constants of the **Value for Associated Standard Action** theme |
| Start a new process | String | 4D_start_new_process | Activate the "Start New Process" option<br>0 = No, 1 = Yes |

## Multistyle Text

| Constant | Type | Value | Comment |
|---|---|---|---|
| ST 4D Expressions as sources | Longint | 2 | The original string of 4D expression references is returned |
| ST 4D Expressions as values | Longint | 1 | 4D expression references are returned in their evaluated form (default functioning in forms) |
| ST End highlight | Longint | -1001 | Designates last character of current text selection in object (*) |
| ST End text | Longint | 0 | Designates last character of text contained in object |
| ST Expression type | Longint | 2 | Selection contains only an expression reference |
| ST Expressions display mode | Longint | 1 | The *value* parameter can contain ST Values or ST References |
| ST Mixed type | Longint | 3 | Selection contains at least two different types of contents |
| ST Picture type | Longint | 6 | Selection contains only a picture (4D Write Pro areas only) |
| ST Plain type | Longint | 0 | Selection contains text and no references |
| ST References | Longint | 1 | Display source strings of expressions |
| ST References as spaces | Longint | 0 | Each reference is returned as a non-breaking space character (default operation, used by other commands) |
| ST Start highlight | Longint | -1000 | Designates first character of current text selection in object (*) |
| ST Start text | Longint | 1 | Designates first character of text contained in object |
| ST Tags as plain text | Longint | 64 | The label of the tag is returned in plain text. For example for the tag '<img src="test.jpg" alt="picture">my picture</img>', the plain text is "my picture" (default functioning in forms) |
| ST Tags as XML code | Longint | 128 | The XML code of the tag is returned in plain text. For example for the tag '<img src="test.jpg" alt="picture">my picture</img>', the plain text is '<img src="test.jpg" alt="picture">my picture</img>' |
| ST Text displayed with 4D Expression sources | Longint | 86 | Returns the text as it is shown in the forms with the original string of the 4D expressions.<br>Corresponds a predefined combination of constants 2+4+16+64. |
| ST Text displayed with 4D Expression values | Longint | 85 | Returns the text as it is shown in the forms with the 4D expressions in their evaluated form.<br>Corresponds to a predefined combination of constants 1+4+16+64. |
| ST Unknown tag type | Longint | 4 | Selection contains only an unknown tag type |
| ST URL as labels | Longint | 4 | The visible label of URLs is returned, for example "Visit our Web site" (default functioning in forms) |
| ST URL as links | Longint | 8 | The link is returned, for example "http://www.4d.com" |
| ST URL type | Longint | 1 | Selection contains only a URL reference |
| ST User links as labels | Longint | 16 | The visible label of the user link is returned (default functioning in forms) |
| ST User links as links | Longint | 32 | The contents of the user link is returned |
| ST User type | Longint | 5 | Selection contains only a custom reference |
| ST Values | Longint | 0 | Display computed values of expressions |

## Multistyle Text Attributes

| Constant | Type | Value | Comment |
|---|---|---|---|
| Attribute background color | Longint | 8 | *attribValue*=Hexadecimal values or HTML color names (Windows only) |
| Attribute bold style | Longint | 1 | *attribValue*=0: remove bold attribute from selection<br>*attribValue*=1: apply bold attribute to selection |
| Attribute font name | Longint | 5 | *attribValue*=Font family name (string) |
| Attribute italic style | Longint | 2 | *attribValue*=0: remove italic attribute from selection<br>*attribValue*=1: apply italic attribute to selection |
| Attribute strikethrough style | Longint | 3 | *attribValue*=0: remove strikethrough attribute from selection<br>*attribValue*=1: apply strikethrough attribute to selection |
| Attribute text color | Longint | 7 | *attribValue*=Hexadecimal values or HTML color names |
| Attribute text size | Longint | 6 | *attribValue*=Number of points (number) |
| Attribute underline style | Longint | 4 | *attribValue*=0: remove underline attribute from selection<br>*attribValue*=1: apply underline attribute to selection |

## Open Form Window

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_Compositing mode form | Longint | 4096 | *** Obsolete constant *** |
| _o_Has toolbar button Mac OS | Longint | 8192 | *** Obsolete constant *** |
| At the bottom | Longint | 393216 | |
| At the top | Longint | 327680 | |
| Form has full screen mode Mac | Longint | 65536 | |
| Horizontally centered | Longint | 65536 | |
| Modal form dialog box | Longint | 1 | |
| Movable form dialog box | Longint | 5 | |
| On the left | Longint | 131072 | |
| On the right | Longint | 196608 | |
| Palette form window | Longint | 1984 | |
| Plain form window | Longint | 8 | |
| Pop up form window | Longint | 32 | |
| Sheet form window | Longint | 33 | |
| Toolbar form window | Longint | 35 | |
| Vertically centered | Longint | 262144 | |

## Open Window

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_Compositing Mode | Longint | 4096 | *** Obsolete constant *** |
| _o_Has toolbar button Mac | Longint | 8192 | *** Obsolete constant *** |
| Alternate dialog box | Longint | 3 | Can be a floating window |
| Has full screen mode Mac | Longint | 65536 | |
| Has grow box | Longint | 4 | |
| Has highlight | Longint | 1 | |
| Has window title | Longint | 2 | |
| Has zoom box | Longint | 8 | |
| Modal dialog box | Longint | 1 | |
| Movable dialog box | Longint | 5 | Can be a floating window |
| Palette window | Longint | 1984 | Can be a floating window |
| Plain dialog box | Longint | 2 | Can be a floating window |
| Plain fixed size window | Longint | 4 | |
| Plain no zoom box window | Longint | 0 | |
| Plain window | Longint | 8 | |
| Pop up window | Longint | 32 | |
| Resizable sheet window | Longint | 34 | |
| Round corner window | Longint | 16 | |
| Sheet window | Longint | 33 | |
| Texture appearance | Longint | 2048 | |

## Pasteboard

| Constant | Type | Value | Comment |
|---|---|---|---|
| No such data in pasteboard | Longint | -102 | |
| Picture data | String | PICT | |
| Text data | String | TEXT | |

## PHP

| Constant | Type | Value | Comment |
|---|---|---|---|
| PHP privileges | Longint | 1 | Definition of specific user privileges relating to the execution of the script.<br>**Possible value(s)**: String in the form "User:Password". For example: "root:jd51254d" |
| PHP raw result | Longint | 2 | Definition of processing mode for HTTP headers returned by PHP in the execution result when this result is of the Text type (when the result is of the BLOB type, headers are always kept).<br>**Possible value(s)**: Boolean. False (default value = remove HTTP headers from result. True = keep HTTP headers. |

## Picture Compression

| Constant | Type | Value | Comment |
|---|---|---|---|
| QT animation compressor | String | rle | |
| QT compact video compressor | String | cdvc | |
| QT graphics compressor | String | smc | |
| QT photo compressor | String | jpeg | |
| QT raw compressor | String | raw | |
| QT video compressor | String | rpza | |

## Picture Display Formats

| Constant | Type | Value | Comment |
|---|---|---|---|
| On background | Longint | 3 | |
| Replicated | Longint | 7 | |
| Scaled to fit | Longint | 2 | |
| Scaled to fit prop centered | Longint | 6 | |
| Scaled to fit proportional | Longint | 5 | |
| Truncated centered | Longint | 1 | |
| Truncated non centered | Longint | 4 | |

## Picture Metadata Names

| Constant | Type | Value | Comment |
|---|---|---|---|
| EXIF aperture value | String | EXIF/ApertureValue | Possible values: Real (APEX value) |
| EXIF brightness value | String | EXIF/BrightnessValue | Possible values: Real (APEX value) |
| EXIF color space | String | EXIF/ColorSpace | Possible values: EXIF Adobe RGB, EXIF s RGB, EXIF Uncalibrated<br>(read only) |
| EXIF components configuration | String | EXIF/ComponentsConfiguration | Possible values: EXIF B, EXIF Cb, EXIF Cr, EXIF G, EXIF R, EXIF Unused, EXIF Y |
| EXIF compressed bits per pixel | String | EXIF/CompressedBitsPerPixel | Possible values: Real |
| EXIF contrast | String | EXIF/Contrast | Possible values: EXIF High, EXIF Low, EXIF Normal |
| EXIF custom rendered | String | EXIF/CustomRendered | Possible values: EXIF Normal, EXIF Custom |
| EXIF date time digitized | String | EXIF/DateTimeDigitized | Possible values: Date or Text (XML Datetime) |
| EXIF date time original | String | EXIF/DateTimeOriginal | Possible values: Date or Text (XML Datetime) |
| EXIF digital zoom ratio | String | EXIF/DigitalZoomRatio | Possible values: Real |
| EXIF EXIF version | String | EXIF/ExifVersion | Possible values: Integer array (4 values) |
| EXIF exposure bias value | String | EXIF/ExposureBiasValue | Possible values: Real |
| EXIF exposure index | String | EXIF/ExposureIndex | Possible values: Real |
| EXIF exposure mode | String | EXIF/ExposureModus | Possible values: EXIF Auto, EXIF Auto Bracket, EXIF Manual |
| EXIF exposure program | String | EXIF/ExposureProgram | Possible values: EXIF Manual, EXIF Action, EXIF Aperture Priority AE, EXIF Creative, EXIF Landscape, EXIF Exposure Portrait, EXIF Program AE, EXIF Shutter Speed Priority AE |
| EXIF exposure time | String | EXIF/ExposureTime | Possible values: Real |
| EXIF F number | String | EXIF/FNumber | Possible values: Real |
| EXIF file source | String | EXIF/FileSource | Possible values: EXIF Digital Camera, EXIF Film Scanner, EXIF Reflection Print Scanner |
| EXIF flash | String | EXIF/Flash | Possible values: EXIF Auto Mode, EXIF Compulsory Flash Firing, EXIF Compulsory Flash Suppression, EXIF Unknown, EXIF Detected, EXIF No Detection Function, EXIF Not Detected, EXIF Reserved |
| EXIF flash energy | String | EXIF/FlashEnergy | Possible values: Real |
| EXIF flash fired | String | EXIF/Flash/Fired | Possible values: Boolean |
| EXIF flash function present | String | EXIF/Flash/FunctionPresent | Possible values: Boolean |

| Constant | Type | Value | Comment |
|---|---|---|---|
| EXIF flash mode | String | EXIF/Flash/Mode | Possible values: EXIF Auto Mode, EXIF Compulsory Flash Firing, EXIF Compulsory Flash Suppression, EXIF Unknown |
| EXIF flash pix version | String | EXIF/FlashPixVersion | Possible values: Integer array (4 values) |
| EXIF flash red eye reduction | String | EXIF/Flash/RedEyeReduction | Possible values: Boolean |
| EXIF flash return light | String | EXIF/Flash/ReturnLight | Possible values: EXIF Detected, EXIF No Detection Function, EXIF Not Detected, EXIF Reserved |
| EXIF focal len in 35 mm film | String | EXIF/FocalLenIn35mmFilm | Possible values: Longint |
| EXIF focal length | String | EXIF/FocalLength | Possible values: Real |
| EXIF focal plane resolution unit | String | EXIF/FocalPlaneResolutionUnit | Possible values: Longint |
| EXIF focal plane X resolution | String | EXIF/FocalPlaneXResolution | Possible values: Real |
| EXIF focal plane Y resolution | String | EXIF/FocalPlaneYResolution | Possible values: Real |
| EXIF gain control | String | EXIF/GainControl | Possible values: EXIF High Gain Down, EXIF High Gain Up, EXIF Low Gain Down, EXIF Low Gain Up, EXIF None |
| EXIF gamma | String | EXIF/Gamma | Possible values: Real |
| EXIF image unique ID | String | EXIF/ImageUniqueID | Possible values: Text |
| EXIF ISO speed ratings | String | EXIF/ISOSpeedRatings | Possible values: Longint or Longint array |
| EXIF light source | String | EXIF/LightSource | Possible values: EXIF Unknown, EXIF Cloudy, EXIF Cool White Fluorescent, EXIF D50, EXIF D55, EXIF D65, EXIF D75, EXIF Daylight, EXIF Daylight Fluorescent, EXIF Day White Fluorescent, EXIF Fine Weather, EXIF Flash, EXIF Light Fluorescent, EXIF ISOStudio Tungsten, EXIF Other, EXIF Shade, EXIF Standard Light A, EXIF Standard Light B, EXIF Standard Light C, EXIF Tungsten, EXIF White Fluorescent |
| EXIF maker note | String | EXIF/MakerNote | Possible values: Text |
| EXIF max aperture value | String | EXIF/MaxApertureValue | Possible values: Real |
| EXIF metering mode | String | EXIF/MeteringMode | Possible values: EXIF Other, EXIF Average, EXIF Center Weighted Average, EXIF Multi Segment, EXIF Multi Spot, EXIF Partial, EXIF Spot |
| EXIF pixel X dimension | String | EXIF/PixelXDimension | Possible values: Longint<br>(read only) |
| EXIF pixel Y dimension | String | EXIF/PixelYDimension | Possible values: Longint<br>(read only) |
| EXIF related sound file | String | EXIF/RelatedSoundFile | Possible values: Text |
| EXIF saturation | String | EXIF/Saturation | Possible values: EXIF High, EXIF Low, EXIF Normal |
| EXIF scene capture type | String | EXIF/SceneCaptureType | Possible values: EXIF Scene Landscape, EXIF Night, EXIF Scene Portrait, EXIF Standard |

| Constant | Type | Value | Comment |
|---|---|---|---|
| EXIF scene type | String | EXIF/SceneType | Possible values: Longint |
| EXIF sensing method | String | EXIF/SensingMethod | Possible values: EXIF Color Sequential Area, EXIF Color Sequential Linear, EXIF Not Defined, EXIF One Chip Color Area, EXIF Three Chip Color Area, EXIF Trilinear, EXIF Two Chip Color Area |
| EXIF sharpness | String | EXIF/Sharpness | Possible values: EXIF High, EXIF Low, EXIF Normal |
| EXIF shutter speed value | String | EXIF/ShutterSpeedValue | Possible values: Real |
| EXIF spectral sensitivity | String | EXIF/SpectralSensitivity | Possible values: Text |
| EXIF subject area | String | EXIF/SubjectArea | Possible values: Longint array (2, 3 or 4 values) |
| EXIF subject dist range | String | EXIF/SubjectDistRange | Possible values: EXIF Unknown, EXIF Close, EXIF Distant, EXIF Macro |
| EXIF subject distance | String | EXIF/SubjectDistance | Possible values: Real |
| EXIF subject location | String | EXIF/SubjectLocation | Possible values: Longint array (2 values) |
| EXIF user comment | String | EXIF/UserComment | Possible values: Text |
| EXIF white balance | String | EXIF/WhiteBalance | Possible values: EXIF Auto, EXIF Manual |
| GPS altitude | String | GPS/Altitude | Possible values: GPS Above Sea Level, GPS Below Sea Level |
| GPS altitude ref | String | GPS/AltitudeRef | Possible values: GPS Above Sea Level, GPS Below Sea Level |
| GPS area information | String | GPS/AreaInformation | Possible values: Text |
| GPS date time | String | GPS/DateTime | Possible values: Date or Text (XML Datetime) |
| GPS dest bearing | String | GPS/DestBearing | Possible values: Text (1 character) |
| GPS dest bearing ref | String | GPS/DestBearingRef | Possible values: Text (1 character) |
| GPS dest distance | String | GPS/DestDistance | Possible values: Text (1 character) |
| GPS dest distance ref | String | GPS/DestDistanceRef | Possible values: Text (1 character) |
| GPS dest latitude | String | GPS/DestLatitude | Possible values: Text |
| GPS dest latitude deg | String | GPS/DestLatitude/Deg | Possible values: Real |
| GPS dest latitude dir | String | GPS/DestLatitude/Dir | Possible values: Text (1 character) |
| GPS dest latitude min | String | GPS/DestLatitude/Min | Possible values: Real |
| GPS dest latitude sec | String | GPS/DestLatitude/Sec | Possible values: Real |
| GPS dest longitude | String | GPS/DestLongitude | Possible values: Text |
| GPS dest longitude deg | String | GPS/DestLongitude/Deg | Possible values: Real |
| GPS dest longitude dir | String | GPS/DestLongitude/Dir | Possible values: Text (1 character) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| GPS dest longitude min | String | GPS/DestLongitude/Min | Possible values: Real |
| GPS dest longitude sec | String | GPS/DestLongitude/Sec | Possible values: Real |
| GPS differential | String | GPS/Differential | Possible values: GPS Correction Applied, GPS Correction Not Applied |
| GPS DOP | String | GPS/DOP | Possible values: Real |
| GPS img direction | String | GPS/ImgDirection | Possible values: GPS Magnetic north, GPS True north |
| GPS img direction ref | String | GPS/ImgDirectionRef | Possible values: GPS Magnetic north, GPS True north |
| GPS latitude | String | GPS/Latitude | Possible values: GPS North, GPS South |
| GPS latitude deg | String | GPS/Latitude/Deg | Possible values: Real |
| GPS latitude dir | String | GPS/Latitude/Dir | Possible values: GPS North, GPS South |
| GPS latitude min | String | GPS/Latitude/Min | Possible values: Real |
| GPS latitude sec | String | GPS/Latitude/Sec | Possible values: Real |
| GPS longitude | String | GPS/Longitude | Possible values: GPS West, GPS East |
| GPS longitude deg | String | GPS/Longitude/Deg | Possible values: Real |
| GPS longitude dir | String | GPS/Longitude/Dir | Possible values: GPS West, GPS East |
| GPS longitude min | String | GPS/Longitude/Min | Possible values: Real |
| GPS longitude sec | String | GPS/Longitude/Sec | Possible values: Real |
| GPS map date | String | GPS/MapDate | Possible values: Text |
| GPS measure mode | String | GPS/MeasureMode | Possible values: GPS 2D, GPS 3D |
| GPS processing method | String | GPS/ProcessingMethod | Possible values: Text |
| GPS satellites | String | GPS/Satellites | Possible values: Text |
| GPS speed | String | GPS/Speed | Possible values: GPS km h, GPS miles h, GPS knots h |
| GPS speed ref | String | GPS/SpeedRef | Possible values: GPS km h, GPS miles h, GPS knots h |
| GPS status | String | GPS/Status | Possible values: GPS Measurement in progress, GPS Measurement Interoperability |
| GPS track | String | GPS/Track | Possible values: Real (0.00..359.99) |
| GPS track ref | String | GPS/TrackRef | Possible values: Text (1 character) |
| GPS version ID | String | GPS/VersionID | Possible values: Longint array (4 characters) |
| IPTC byline | String | IPTC/Byline | Possible values: Text or Text array |
| IPTC byline title | String | IPTC/BylineTitle | Possible values: Text or Text array |
| IPTC caption abstract | String | IPTC/CaptionAbstract | Possible values: Text |
| IPTC category | String | IPTC/Category | Possible values: Text |

| Constant | Type | Value | Comment |
|---|---|---|---|
| IPTC city | String | IPTC/City | Possible values: Text |
| IPTC contact | String | IPTC/Contact | Possible values: Text or Text array |
| IPTC content location code | String | IPTC/ContentLocationCode | Possible values: Text or Text array |
| IPTC content location name | String | IPTC/ContentLocationName | Possible values: Text or Text array |
| IPTC copyright notice | String | IPTC/CopyrightNotice | Possible values: Text |
| IPTC country primary location code | String | IPTC/CountryPrimaryLocationCode | Possible values: Text |
| IPTC country primary location name | String | IPTC/CountryPrimaryLocationName | Possible values: Text |
| IPTC credit | String | IPTC/Credit | Possible values: Text |
| IPTC date time created | String | IPTC/DateTimeCreated | Possible values: Date or Text (XML Datetime) |
| IPTC digital creation date time | String | IPTC/DigitalCreationDateTime | Possible values: Date or Text (XML Datetime) |
| IPTC edit status | String | IPTC/EditStatus | Possible values: Text |
| IPTC expiration date time | String | IPTC/ExpirationDateTime | Possible values: Date or Text (XML Datetime) |
| IPTC fixture identifier | String | IPTC/FixtureIdentifier | Possible values: Text |
| IPTC headline | String | IPTC/Headline | Possible values: Text |
| IPTC image orientation | String | IPTC/ImageOrientation | Possible values: Text |
| IPTC image type | String | IPTC/ImageType | Possible values: Text |
| IPTC keywords | String | IPTC/Keywords | Possible values: Text or Text array |
| IPTC language identifier | String | IPTC/LanguageIdentifier | Possible values: Text |
| IPTC object attribute reference | String | IPTC/ObjectAttributeReference | Possible values: Text |
| IPTC object cycle | String | IPTC/ObjectCycle | Possible values: Text |
| IPTC object name | String | IPTC/ObjektName | Possible values: Text |
| IPTC original transmission reference | String | IPTC/OriginalTransmissionReference | Possible values: Text |
| IPTC originating program | String | IPTC/OriginatingProgram | Possible values: Text |
| IPTC program version | String | IPTC/ProgramVersion | Possible values: Text |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| IPTC province state | String | IPTC/ProvinceState | Possible values: Text |
| IPTC release date time | String | IPTC/ReleaseDateTime | Possible values: Date or Text (XML Datetime) |
| IPTC scene | String | IPTC/Scene | Possible values: IPTC Action, IPTC Aerial View, IPTC Close Up, IPTC Couple, IPTC Exterior View, IPTC Full Length, IPTC General View, IPTC Group, IPTC Half Length, IPTC Headshot, IPTC Interior View, IPTC Movie Scene, IPTC Night Scene, IPTC Off Beat, IPTC Panoramic View, IPTC Performing, IPTC Posing, IPTC Profile, IPTC Rear View, IPTC Satellite, IPTC Single, IPTC Symbolic, IPTC Two, IPTC Under Water |
| IPTC source | String | IPTC/Source | Possible values: Text |
| IPTC special instructions | String | IPTC/SpecialInstructions | Possible values: Text |
| IPTC star rating | String | IPTC/StarRating | Possible values: Longint |
| IPTC sub location | String | IPTC/SubLocation | Possible values: Text |
| IPTC subject reference | String | IPTC/SubjectReference | Possible values: Longint or Longint array |
| IPTC supplemental category | String | IPTC/SupplementalCategory | Possible values: Text or Text array |
| IPTC urgency | String | IPTC/Urgency | Possible values: Longint |
| IPTC writer editor | String | IPTC/WriterEditor | Possible values: Text or Text array |
| TIFF artist | String | TIFF/Artist | Possible values: Text |
| TIFF compression | String | TIFF/Compression | Possible values: TIFF Adobe Deflate, TIFF CCIRLEW, TIFF CCITT1D, TIFF DCS, TIFF Deflate, TIFF Epson ERF, TIFF IT8BL, TIFF IT8CTPAD, TIFF IT8LW, TIFF IT8MP, TIFF JBIG, TIFF JBIGB&W, TIFF JBIGColor, TIFF JPEG, TIFF JPEG2000, TIFF JPEGThumbs Only, TIFF Kodak262, TIFF Kodak DCR, TIFF Kodak KDC, TIFF LZW, TIFF MDIBinary Level Codec, TIFF MDIProgressive Transform Codec, TIFF MDIVector, TIFF Next, TIFF Nikon NEF, TIFF Pack Bits, TIFF Pentax PEF, TIFF Pixar Film, TIFF Pixar Log, TIFF SGILog, TIFF SGILog24, TIFF Sony ARW, TIFF T4Group3Fax, TIFF T6Group4Fax, TIFF Thunderscan, TIFF Uncompressed |
| TIFF copyright | String | TIFF/Copyright | Possible values: Text |
| TIFF date time | String | TIFF/DateTime | Possible values: Date or Text (XML Datetime) |
| TIFF document name | String | TIFF/DocumentName | Possible values: Text |
| TIFF host computer | String | TIFF/HostComputer | Possible values: Text |
| TIFF image description | String | TIFF/ImageDescription | Possible values: Text |
| TIFF make | String | TIFF/Make | Possible values: Text |
| TIFF model | String | TIFF/Model | Possible values: Text |
| TIFF orientation | String | TIFF/Orientation | Possible values: TIFF Horizontal, TIFF Mirror Horizontal, TIFF Mirror Horizontal And Rotate270CW, TIFF Mirror Horizontal And Rotate90CW, TIFF Mirror Vertical, TIFF Rotate180, TIFF Rotate270CW, TIFF Rotate90CW |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| TIFF photometric interpretation | String | TIFF/PhotometricInterpretation | Possible values: TIFF Black Is Zero, TIFF CIELab, TIFF CMYK, TIFF Color Filter Array, TIFF ICCLab, TIFF ITULab, TIFF Linear Raw, TIFF Pixar Log L, TIFF Pixar Log Luv, TIFF RGB, TIFF RGBPalette, TIFF Transparency Mask, TIFF White Is Zero, TIFF YCb Cr |
| TIFF resolution unit | String | TIFF/ResolutionUnit | Possible values: TIFF CM, TIFF Inches, TIFF MM, TIFF None, TIFF UM |
| TIFF software | String | TIFF/Software | Possible values: Text |
| TIFF xResolution | String | TIFF/XResolution | Possible values: Real (read only) |
| TIFF yResolution | String | TIFF/YResolution | Possible values: Real (read only) |

## Picture Metadata Values

| Constant | Type | Value | Comment |
|---|---|---|---|
| EXIF Action | Longint | 6 | |
| EXIF Adobe RGB | Longint | 2 | |
| EXIF Aperture Priority AE | Longint | 3 | |
| EXIF Auto | Longint | 0 | |
| EXIF Auto Bracket | Longint | 2 | |
| EXIF Auto Mode | Longint | 3 | |
| EXIF Average | Longint | 1 | |
| EXIF B | Longint | 6 | |
| EXIF Cb | Longint | 2 | |
| EXIF Center Weighted Average | Longint | 2 | |
| EXIF Close | Longint | 2 | |
| EXIF Cloudy | Longint | 10 | |
| EXIF Color Sequential Area | Longint | 5 | |
| EXIF Color Sequential Linear | Longint | 8 | |
| EXIF Compulsory Flash Firing | Longint | 1 | |
| EXIF Compulsory Flash Suppression | Longint | 2 | |
| EXIF Cool White Fluorescent | Longint | 14 | |
| EXIF Cr | Longint | 3 | |
| EXIF Creative | Longint | 5 | |
| EXIF Custom | Longint | 1 | |
| EXIF D50 | Longint | 23 | |
| EXIF D55 | Longint | 20 | |
| EXIF D65 | Longint | 21 | |
| EXIF D75 | Longint | 22 | |
| EXIF Day White Fluorescent | Longint | 13 | |
| EXIF Daylight | Longint | 1 | |
| EXIF Daylight Fluorescent | Longint | 12 | |
| EXIF Detected | Longint | 3 | |
| EXIF Digital Camera | Longint | 3 | |
| EXIF Distant | Longint | 3 | |
| EXIF Exposure Portrait | Longint | 7 | |
| EXIF Film Scanner | Longint | 1 | |
| EXIF Fine Weather | Longint | 9 | |
| EXIF Flashlight | Longint | 4 | |
| EXIF G | Longint | 5 | |
| EXIF High | Longint | 2 | |
| EXIF High Gain Down | Longint | 4 | |
| EXIF High Gain Up | Longint | 2 | |
| EXIF ISOStudio Tungsten | Longint | 24 | |
| EXIF Landscape | Longint | 8 | |
| EXIF Light Fluorescent | Longint | 2 | |
| EXIF Low | Longint | 1 | |
| EXIF Low Gain Down | Longint | 3 | |
| EXIF Low Gain Up | Longint | 1 | |
| EXIF Macro | Longint | 1 | |
| EXIF Manual | Longint | 1 | |
| EXIF Multi Segment | Longint | 5 | |
| EXIF Multi Spot | Longint | 4 | |
| EXIF Night | Longint | 3 | |
| EXIF No Detection Function | Longint | 0 | |
| EXIF None | Longint | 0 | |

| Constant | Type | Value | Comment |
|---|---|---|---|
| EXIF Normal | Longint | 0 | |
| EXIF Not Defined | Longint | 1 | |
| EXIF Not Detected | Longint | 2 | |
| EXIF One Chip Color Area | Longint | 2 | |
| EXIF Other | Longint | 255 | |
| EXIF Partial | Longint | 6 | |
| EXIF Program AE | Longint | 2 | |
| EXIF R | Longint | 4 | |
| EXIF Reflection Print Scanner | Longint | 2 | |
| EXIF Reserved | Longint | 1 | |
| EXIF s RGB | Longint | 1 | |
| EXIF Scene Landscape | Longint | 1 | |
| EXIF Scene Portrait | Longint | 2 | |
| EXIF Shade | Longint | 11 | |
| EXIF Shutter Speed Priority AE | Longint | 4 | |
| EXIF Spot | Longint | 3 | |
| EXIF Standard | Longint | 0 | |
| EXIF Standard Light A | Longint | 17 | |
| EXIF Standard Light B | Longint | 18 | |
| EXIF Standard Light C | Longint | 19 | |
| EXIF Three Chip Color Area | Longint | 4 | |
| EXIF Trilinear | Longint | 7 | |
| EXIF Tungsten | Longint | 3 | |
| EXIF Two Chip Color Area | Longint | 3 | |
| EXIF Uncalibrated | Longint | -1 | |
| EXIF Unknown | Longint | 0 | |
| EXIF Unused | Longint | 0 | |
| EXIF White Fluorescent | Longint | 15 | |
| EXIF Y | Longint | 1 | |
| GPS 2D | Longint | 2 | |
| GPS 3D | Longint | 3 | |
| GPS Above Sea Level | Longint | 0 | |
| GPS Below Sea Level | Longint | 1 | |
| GPS Correction Applied | Longint | 1 | |
| GPS Correction Not Applied | Longint | 0 | |
| GPS East | String | E | |
| GPS km h | String | K | |
| GPS knots h | String | K | |
| GPS Magnetic north | String | M | |
| GPS Measurement in progress | String | A | |
| GPS Measurement Interoperability | String | V | |
| GPS miles h | String | M | |
| GPS North | String | N | |
| GPS South | String | S | |
| GPS True north | String | T | |
| GPS West | String | W | |
| IPTC Action | Longint | 11900 | |
| IPTC Aerial View | Longint | 11200 | |
| IPTC Close Up | Longint | 11800 | |
| IPTC Couple | Longint | 10700 | |
| IPTC Exterior View | Longint | 11600 | |

| Constant | Type | Value | Comment |
|---|---|---|---|
| IPTC Full Length | Longint | 10300 | |
| IPTC General View | Longint | 11000 | |
| IPTC Group | Longint | 10900 | |
| IPTC Half Length | Longint | 10200 | |
| IPTC Headshot | Longint | 10100 | |
| IPTC Interior View | Longint | 11700 | |
| IPTC Movie Scene | Longint | 12400 | |
| IPTC Night Scene | Longint | 11400 | |
| IPTC Off Beat | Longint | 12300 | |
| IPTC Panoramic View | Longint | 11100 | |
| IPTC Performing | Longint | 12000 | |
| IPTC Posing | Longint | 12100 | |
| IPTC Profile | Longint | 10400 | |
| IPTC Rear View | Longint | 10500 | |
| IPTC Satellite | Longint | 11500 | |
| IPTC Single | Longint | 10600 | |
| IPTC Symbolic | Longint | 12200 | |
| IPTC Two | Longint | 10800 | |
| IPTC Under Water | Longint | 11300 | |
| TIFF Adobe Deflate | Longint | 8 | |
| TIFF Black Is Zero | Longint | 1 | |
| TIFF CCIRLEW | Longint | 32771 | |
| TIFF CCITT1D | Longint | 2 | |
| TIFF CIELab | Longint | 8 | |
| TIFF CM | Longint | 3 | |
| TIFF CMYK | Longint | 5 | |
| TIFF Color Filter Array | Longint | 32803 | |
| TIFF DCS | Longint | 32947 | |
| TIFF Deflate | Longint | 32946 | |
| TIFF Epson ERF | Longint | 32769 | |
| TIFF Horizontal | Longint | 1 | |
| TIFF ICCLab | Longint | 9 | |
| TIFF Inches | Longint | 2 | |
| TIFF IT8BL | Longint | 32898 | |
| TIFF IT8CTPAD | Longint | 32895 | |
| TIFF IT8LW | Longint | 32896 | |
| TIFF IT8MP | Longint | 32897 | |
| TIFF ITULab | Longint | 10 | |
| TIFF JBIG | Longint | 34661 | |
| TIFF JBIGB&W | Longint | 9 | |
| TIFF JBIGColor | Longint | 10 | |
| TIFF JPEG | Longint | 7 | |
| TIFF JPEG2000 | Longint | 34712 | |
| TIFF JPEGThumbs Only | Longint | 6 | |
| TIFF Kodak DCR | Longint | 65000 | |
| TIFF Kodak KDC | Longint | 32867 | |
| TIFF Kodak262 | Longint | 262 | |
| TIFF Linear Raw | Longint | 34892 | |
| TIFF LZW | Longint | 5 | |
| TIFF MDIBinary Level Codec | Longint | 34718 | |
| TIFF MDIProgressive Transform Codec | Longint | 34719 | |

| Constant | Type | Value | Comment |
|---|---|---|---|
| TIFF MDIVector | Longint | 34720 | |
| TIFF Mirror Horizontal | Longint | 2 | |
| TIFF Mirror Horizontal And Rotate270CW | Longint | 5 | |
| TIFF Mirror Horizontal And Rotate90CW | Longint | 7 | |
| TIFF Mirror Vertical | Longint | 4 | |
| TIFF MM | Longint | 4 | |
| TIFF Next | Longint | 32766 | |
| TIFF Nikon NEF | Longint | 34713 | |
| TIFF None | Longint | 1 | |
| TIFF Pack Bits | Longint | 32773 | |
| TIFF Pentax PEF | Longint | 65535 | |
| TIFF Pixar Film | Longint | 32908 | |
| TIFF Pixar Log | Longint | 32909 | |
| TIFF Pixar Log L | Longint | 32844 | |
| TIFF Pixar Log Luv | Longint | 32845 | |
| TIFF RGB | Longint | 2 | |
| TIFF RGBPalette | Longint | 3 | |
| TIFF Rotate180 | Longint | 3 | |
| TIFF Rotate270CW | Longint | 8 | |
| TIFF Rotate90CW | Longint | 6 | |
| TIFF SGILog | Longint | 34676 | |
| TIFF SGILog24 | Longint | 34677 | |
| TIFF Sony ARW | Longint | 32767 | |
| TIFF T4Group3Fax | Longint | 3 | |
| TIFF T6Group4Fax | Longint | 4 | |
| TIFF Thunderscan | Longint | 32809 | |
| TIFF Transparency Mask | Longint | 4 | |
| TIFF UM | Longint | 5 | |
| TIFF Uncompressed | Longint | 1 | |
| TIFF White Is Zero | Longint | 0 | |
| TIFF YCb Cr | Longint | 6 | |

## Picture Transformation

| Constant | Type | Value | Comment |
|---|---|---|---|
| Crop | Longint | 100 | |
| Fade to grey scale | Longint | 101 | |
| Flip horizontally | Longint | 3 | |
| Flip vertically | Longint | 4 | |
| Horizontal concatenation | Longint | 1 | |
| Reset | Longint | 0 | |
| Scale | Longint | 1 | |
| Superimposition | Longint | 3 | |
| Translate | Longint | 2 | |
| Transparency | Longint | 102 | |
| Vertical concatenation | Longint | 2 | |

## Platform Interface

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Automatic platform | Longint | -1 | |
| Mac OS 7 | Longint | 0 | |
| Mac OS 9 | Longint | 3 | |
| Mac theme | Longint | 4 | |
| Windows 3.11, NT 3.51 | Longint | 1 | |
| Windows 9x | Longint | 2 | |

## Platform Properties

Constants prefixed with _o_ are obsolete and cannot be returned by the command. They are only kept for compatibility.

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_INTEL 386 | Longint | 386 | |
| _o_INTEL 486 | Longint | 486 | |
| _o_Macintosh 68K | Longint | 1 | |
| _o_PowerPC 601 | Longint | 601 | |
| _o_PowerPC 603 | Longint | 603 | |
| _o_PowerPC 604 | Longint | 604 | |
| _o_PowerPC G3 | Longint | 510 | |
| Intel compatible | Longint | 586 | |
| Mac OS | Longint | 2 | |
| Power PC | Longint | 406 | |
| Windows | Longint | 3 | |

## Print Options

| Constant | Type | Value | Comment |
|---|---|---|---|
| Color option | Longint | 8 | (Windows only) *value1* only: code specifying the mode for handling color: 1=Black and white (monochrome), 2=Color.<br>**64-bit versions**: This option is not supported in 4D 64-bit versions (obsolete) |
| Destination option | Longint | 9 | *value1*: code specifying the type of print destination: 1=Printer, 2=(PC)/PS File (Mac), 3=PDF file, 5=Screen (OS X driver option).<br>If *value1* is different from 1 or 5, *value2* contains pathname for resulting document. This path will be used until another path is specified. If a file with the same name already exists at the destination location, it will be replaced. With **GET PRINT OPTION**, if the current value is not in the predefined list, *value1* contains -1 and the system variable OK is set to 1. If an error occurs, *value1* and the system variable OK are set to 0.<br>**Note**: Under Windows, you can set the printing destination to 3 (PDF File) when the PDF Creator driver has been installed. When the *(9;3;path)* values are passed, 4D automatically starts a "silent" PDF printing which takes into account any option codes that are passed (note that if you pass an empty string in *value2* or omit this parameter, a file saving dialog appears at the time of printing.) After printing, the current settings are restored. This simplifies control of printing PDFs for 4D and lets you write multi-platform code. When the *(9;3;path)* values are not passed, printing is not controlled by 4D and any PDF Creator option codes that were passed are ignored. |
| Double sided option | Longint | 11 | (Windows only) *value1*: 0=Single-sided or standard, 1=Double-sided. If *value1*=1, *value2* contains the binding: 0=Left binding (default value), 1=Top binding.<br>**Note:** This option can only be used under Windows. |
| Generic PDF driver | String | _4d_pdf_printer | **Note:** This function is not available in 32-bit versions of 4D.<br>• On OS X, sets the current printer to the default driver. This driver is invisible; it is not found in the list returned by **PRINTERS LIST**. Note that a PDF document path must have been set using **SET PRINT OPTION**, otherwise error 3107 is returned.<br>• On Windows, sets the current printer to the Windows PDF driver (named "Microsoft Print to PDF"). This constant is only available in Windows 10 with the PDF option installed. In other Windows versions, or if no PDF driver is available, the command does nothing and the *OK* variable is set to 0. |
| Hide printing progress option | Longint | 14 | *value1* only: 1=hide progress windows, 0=display progress windows (default). This option is particularly useful in the case of PDF printing under OS X.<br>**Note:** There is already a Printing progress option found in the Database Settings dialog box (Interface page). However, it is applied globally to the application and does not hide all the windows under OS X. |
| Legacy printing layer option | Longint | 16 | (4D 64-bit versions for Windows only) *value1* only: 1=select the GDI-based legacy printing layer for the subsequent printing jobs. 0=select the D2D printing layer (default).<br>**64-bit versions:** This selector is only supported on 4D 64-bit single-user applications on Windows; it is ignored on other platforms. It is mainly intended to allow legacy plug-ins to print inside 4D jobs in 4D 64-bit applications. |
| Mac spool file format option | Longint | 13 | (Mac only) *value1* only: 0=print job in PDF mode (default value) 1=print job in PostScript mode.<br>**Notes:**<br>- This option has no effect under Windows.<br>- Under OS X, printing is done as a PDF by default. However, the PDF print driver does not support PICT pictures with encapsulated PostScript information — these pictures are generated, more particularly, by vectorial drawing software. To avoid this problem, this option lets you modify the print mode to use under OS X for the current session. Keep in mind that printing in PostScript mode can lead to undesired side effects.<br>**64-bit versions:** This option is not supported; it is replaced by the Generic PDF driver option of the **SET CURRENT PRINTER** command. |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Number of copies option | Longint | 4 | *value1* only: number of copies to be printed. |
| Orientation option | Longint | 2 | *value1* only: 1=Portrait, 2=Landscape. If a different orientation option is used, **GET PRINT OPTION** returns 0 in *value1*.<br>**64-bit versions**: This option can be called within a print job, which means that you can switch from portrait to landscape, or vice versa, during the same print job. |
| Page range option | Longint | 15 | *value1*=first page to print (default value is 1) and (optional) *value2*=number of the last page to print (default value -1 = end of document). |
| Page setup dialog | Longint | 1 | Display the Page setup dialog box |
| Paper option | Longint | 1 | If you use only *value1*, it contains the name of the paper. If you use both parameters, *value1* contains the paper width and *value2* contains the paper height. The width and height are expressed in screen pixels. Use the **PRINT OPTION VALUES** command to get the name, height and width of all the paper formats offered by the printer. |
| Paper source option | Longint | 5 | (Windows only) *value1* only: number corresponding to the index, in the array of trays returned by the **PRINT OPTION VALUES** command, of the paper tray to be used. This option can only be used under Windows. |
| PDFCreator Printer name | String | PDFCreator | |
| Print dialog | Longint | 2 | Display the Print dialog box |
| Scale option | Longint | 3 | *value1* only: scale value in percentage. Be careful, some printers do not allow you to modify the scale. If you pass an invalid value, the property is reset to 100% at the time of printing. |
| Spooler document name option | Longint | 12 | *value1* only: name of the current print document, which appears in the list of spooler documents. The name defined by this statement will be used for all the print documents of the session for as long as a new name or an empty string is not passed. To use or restore standard operation (using the method name in the case of a method, the table name for a record, etc.), pass an empty string in *value1*. |

## Process State

| Constant | Type | Value | Comment |
|---|---|---|---|
| Aborted | Longint | -1 | |
| Delayed | Longint | 1 | |
| Does not exist | Longint | -100 | |
| Executing | Longint | 0 | |
| Hidden modal dialog | Longint | 6 | |
| Paused | Longint | 5 | |
| Waiting for input output | Longint | 3 | |
| Waiting for internal flag | Longint | 4 | |
| Waiting for user event | Longint | 2 | |

## Process Type

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_Web process with context | Longint | -11 | |
| Apple event manager | Longint | -7 | |
| Backup process | Longint | -19 | |
| Cache manager | Longint | -4 | |
| Client manager process | Longint | -31 | |
| Created from execution dialog | Longint | 3 | |
| Created from menu command | Longint | 2 | |
| Design process | Longint | -2 | |
| Event manager | Longint | -8 | |
| Execute on client process | Longint | -14 | |
| Execute on server process | Longint | 1 | |
| External task | Longint | -9 | |
| Indexing process | Longint | -5 | |
| Internal 4D server process | Longint | -18 | |
| Internal timer process | Longint | -25 | |
| Log file process | Longint | -20 | |
| Main process | Longint | -1 | |
| Method editor macro process | Longint | -17 | |
| Monitor process | Longint | -26 | |
| MSC process | Longint | -22 | |
| None | Longint | 0 | |
| On exit process | Longint | -16 | |
| Other 4D process | Longint | -10 | |
| Other user process | Longint | 4 | |
| Restore Process | Longint | -21 | |
| Serial Port Manager | Longint | -6 | |
| Server interface process | Longint | -15 | |
| SQL Method execution process | Longint | -24 | |
| Web process on 4D remote | Longint | -12 | |
| Web process with no context | Longint | -3 | |
| Web server process | Longint | -13 | |
| Worker process | Longint | 5 | Worker process launched by user |

## QR Area Properties

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr view color toolbar | Longint | 5 | Display status of the Color toolbar (Displayed=1, Hidden=0) |
| qr view column toolbar | Longint | 6 | Display status of the Column toolbar (Displayed=1, Hidden=0) |
| qr view contextual menus | Longint | 7 | Display status of the Contextual menu (Displayed=1, Hidden=0) |
| qr view menubar | Longint | 1 | Display status of the menu bar (Displayed=1, Hidden=0) |
| qr view operators toolbar | Longint | 4 | Display status of the Operators toolbar (Displayed=1, Hidden=0) |
| qr view standard toolbar | Longint | 2 | Display status of the Standard toolbar (Displayed=1, Hidden=0) |
| qr view style toolbar | Longint | 3 | Display status of the Style toolbar (Displayed=1, Hidden=0) |

## QR Borders

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr bottom border | Longint | 8 | Bottom border |
| qr inside horizontal border | Longint | 32 | Inside horizontal border |
| qr inside vertical border | Longint | 16 | Inside vertical border |
| qr left border | Longint | 1 | Left border |
| qr right border | Longint | 4 | Right border |
| qr top border | Longint | 2 | Top border |

## QR Commands

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr cmd 4D View destination | Longint | 2503 | |
| qr cmd add column | Longint | 2608 | |
| qr cmd alt back color palette | Longint | 1004 | |
| qr cmd automatic width | Longint | 2605 | |
| qr cmd average | Longint | 507 | |
| qr cmd back color palette | Longint | 1003 | |
| qr cmd back colors toolbar | Longint | 2052 | |
| qr cmd bold | Longint | 500 | |
| qr cmd borders | Longint | 2609 | |
| qr cmd center justified | Longint | 504 | |
| qr cmd columns toolbar | Longint | 2054 | |
| qr cmd count | Longint | 510 | |
| qr cmd default justified | Longint | 512 | |
| qr cmd delete column | Longint | 2601 | |
| qr cmd disk file destination | Longint | 2501 | |
| qr cmd edit column | Longint | 2603 | |
| qr cmd font color palette | Longint | 1002 | |
| qr cmd font dropdown | Longint | 1000 | |
| qr cmd format | Longint | 2606 | |
| qr cmd generate | Longint | 2008 | |
| qr cmd graph destination | Longint | 2502 | |
| qr cmd header and footer | Longint | 2005 | |
| qr cmd hide column | Longint | 2602 | |
| qr cmd hide line | Longint | 2607 | |
| qr cmd HTML file destination | Longint | 2504 | |
| qr cmd insert column | Longint | 2600 | |
| qr cmd italic | Longint | 501 | |
| qr cmd left justified | Longint | 503 | |
| qr cmd max | Longint | 509 | |
| qr cmd min | Longint | 508 | |
| qr cmd move left | Longint | 3002 | |
| qr cmd move right | Longint | 3003 | |
| qr cmd new | Longint | 2000 | |
| qr cmd open | Longint | 2001 | |
| qr cmd operators toolbar | Longint | 2051 | |
| qr cmd page setup | Longint | 2006 | |
| qr cmd plain | Longint | 511 | |
| qr cmd presentation | Longint | 2611 | |
| qr cmd print preview | Longint | 2007 | |
| qr cmd printer destination | Longint | 2500 | |
| qr cmd repeated values | Longint | 2604 | |
| qr cmd revert to save | Longint | 2004 | |
| qr cmd right justified | Longint | 505 | |
| qr cmd save | Longint | 2002 | |
| qr cmd save as | Longint | 2003 | |
| qr cmd standard deviation | Longint | 513 | |
| qr cmd standard toolbar | Longint | 2053 | |
| qr cmd style toolbar | Longint | 2050 | |
| qr cmd sum | Longint | 506 | |
| qr cmd totals spacing | Longint | 2610 | |
| qr cmd underline | Longint | 502 | |

## QR Document Properties

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr printing dialog | Longint | 1 | Display of the print dialog box:<br>• If value = 0, the print dialog is not displayed prior to printing.<br>• If value = 1, the print dialog is displayed prior to printing (default value). |
| qr unit | Longint | 2 | Document unit:<br>• If value = 0, the document unit is points.<br>• If value = 1, the document unit is centimeters.<br>• If value = 2, the document unit is inches. |

## QR Operators

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| qr average | Longint | 2 | |
| qr count | Longint | 16 | |
| qr max | Longint | 8 | |
| qr min | Longint | 4 | |
| qr standard deviation | Longint | 32 | |
| qr sum | Longint | 1 | |

## QR Output Destination

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_qr 4D Chart area | Longint | 4 | *** Obsolete constant *** |
| qr 4D View area | Longint | 3 | *specifics*: N.A. |
| qr HTML file | Longint | 5 | *specifics*: Pathname to the file. |
| qr printer | Longint | 1 | *specifics*: "*" to remove the print dialog boxes |
| qr text file | Longint | 2 | *specifics*: Pathname to the file. |

## QR Report Types

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| qr cross report | Longint | 2 | |
| qr list report | Longint | 1 | |

## QR Rows for Properties

| Constant | Type | Value | Comment |
|---|---|---|---|
| qr detail | Longint | -2 | Detail area of report |
| qr footer | Longint | -5 | Page footer |
| qr grand total | Longint | -3 | Grand total area |
| qr header | Longint | -4 | Page header |
| qr title | Longint | -1 | Title of report |

## QR Text Properties

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_qr font | Longint | 1 | Obsolete since 4D v14R3 (use qr font name) |
| qr alternate background color | Longint | 9 | Alternate background color number |
| qr background color | Longint | 8 | Background color number |
| qr bold | Longint | 3 | Bold style attribute (0 or 1) |
| qr font name | Longint | 10 | Name of font as returned for example by the **FONT LIST** command |
| qr font size | Longint | 2 | Font size expressed in points (9 to 255) |
| qr italic | Longint | 4 | Italic style attribute (0 or 1) |
| qr justification | Longint | 7 | Justification attribute (0 for default, 1 for left, 2 for center or 3 for right) |
| qr text color | Longint | 6 | Color number attribute (Longint) |
| qr underline | Longint | 5 | Underline style attribute (0 or 1) |

## Queries

| Constant | Type | Value | Comment |
|---|---|---|---|
| Description in text format | Longint | 0 | |
| Description in XML format | Longint | 1 | |
| Into current selection | Longint | 0 | |
| Into named selection | Longint | 2 | |
| Into set | Longint | 1 | |
| Into variable | Longint | 3 | |

## Relations

| Constant | Type | Value | Comment |
|---|---|---|---|
| Automatic | Longint | 3 | |
| Do not modify | Longint | 0 | |
| Manual | Longint | 2 | |
| No relation | Longint | 0 | |
| Structure configuration | Longint | 1 | |

## Resources Properties

| Constant | Type | Value | Comment |
|---|---|---|---|
| Changed resource bit | Longint | 1 | |
| Changed resource mask | Longint | 2 | |
| Locked resource bit | Longint | 4 | |
| Locked resource mask | Longint | 16 | |
| Preloaded resource bit | Longint | 2 | |
| Preloaded resource mask | Longint | 4 | |
| Protected resource bit | Longint | 3 | |
| Protected resource mask | Longint | 8 | |
| Purgeable resource bit | Longint | 5 | |
| Purgeable resource mask | Longint | 32 | |
| System heap resource bit | Longint | 6 | |
| System heap resource mask | Longint | 64 | |

## SCREEN DEPTH

| Constant | Type | Value | Comment |
|---|---|---|---|
| Black and white | Longint | 0 | |
| Four colors | Longint | 2 | |
| Is color | Longint | 1 | |
| Is gray scale | Longint | 0 | |
| Millions of colors 24 bit | Longint | 24 | |
| Millions of colors 32 bit | Longint | 32 | |
| Sixteen colors | Longint | 4 | |
| Thousands of colors | Longint | 16 | |
| Two fifty six colors | Longint | 8 | |

## SET RGB COLORS

| Constant | Type | Value | Comment |
|---|---|---|---|
| Background color | Longint | -2 | |
| Background color none | Longint | -16 | This constant can only be used with the *backgroundColor* and *altBackgrndColor* parameters. |
| Dark shadow color | Longint | -3 | |
| Disable highlight item color | Longint | -11 | |
| Foreground color | Longint | -1 | |
| Highlight menu background color | Longint | -9 | |
| Highlight menu text color | Longint | -10 | |
| Highlight text background color | Longint | -7 | |
| Highlight text color | Longint | -8 | |
| Light shadow color | Longint | -4 | |

## Shortcut and Associated Keys

| Constant | Type | Value | Comment |
|---|---|---|---|
| Shortcut with Backspace | String | [backspace] | |
| Shortcut with Carriage Return | String | [return] | |
| Shortcut with Delete | String | [del] | |
| Shortcut with Down arrow | String | [down arrow] | |
| Shortcut with End | String | [end] | |
| Shortcut with Enter | String | [enter] | |
| Shortcut with Escape | String | [esc] | |
| Shortcut with F1 | String | [F1] | |
| Shortcut with F10 | String | [F10] | |
| Shortcut with F11 | String | [F11] | |
| Shortcut with F12 | String | [F12] | |
| Shortcut with F13 | String | [F13] | |
| Shortcut with F14 | String | [F14] | |
| Shortcut with F15 | String | [F15] | |
| Shortcut with F2 | String | [F2] | |
| Shortcut with F3 | String | [F3] | |
| Shortcut with F4 | String | [F4] | |
| Shortcut with F5 | String | [F5] | |
| Shortcut with F6 | String | [F6] | |
| Shortcut with F7 | String | [F7] | |
| Shortcut with F8 | String | [F8] | |
| Shortcut with F9 | String | [F9] | |
| Shortcut with Help | String | [help] | |
| Shortcut with Home | String | [home] | |
| Shortcut with Left arrow | String | [left arrow] | |
| Shortcut with Page down | String | [page down] | |
| Shortcut with Page up | String | [page up] | |
| Shortcut with Right arrow | String | [right arrow] | |
| Shortcut with Tabulation | String | [tab] | |
| Shortcut with Up arrow | String | [up arrow] | |

## SQL

| Constant | Type | Value | Comment |
|---|---|---|---|
| SQL all records | Longint | -1 | |
| SQL asynchronous | Longint | 1 | 0 = Synchronous connection (default value), 1 (or value other than 0) = Asynchronous connection |
| SQL charset | Longint | 100 | Text encoding used for requests sent to external sources (via the SQL pass-through). The modification is carried out for the current process and the current connection.<br>Possible values: MIBEnum identifier (see note 2) or value -2 (see note 3)<br>By default: 106 (UTF-8) |
| SQL connection timeout | Longint | 5 | Maximum timeout awaiting response when executing the **SQL LOGIN** command. This value must be set before opening the connection in order to be taken into account<br>Possible values: time in seconds<br>By default: no timeout |
| SQL max data length | Longint | 3 | Maximum length of data returned |
| SQL max rows | Longint | 2 | Maximum number of rows in resulting group (used for previews) |
| SQL On error abort | Longint | 1 | In the event of an error, 4D immediately stops script execution. |
| SQL On error confirm | Longint | 2 | In the event of an error, 4D displays a dialog box describing the error and allowing the user to interrupt or continue script execution. |
| SQL On error continue | Longint | 3 | In the event of an error, 4D ignores it and continues script execution. |
| SQL param in | Longint | 1 | |
| SQL param in out | Longint | 2 | Usable only in the context of an SQL stored procedure (in-out parameter defined in the stored procedure) |
| SQL param out | Longint | 4 | Usable only in the context of an SQL stored procedure (out parameter defined in the stored procedure) |
| SQL query timeout | Longint | 4 | Maximum timeout awaiting response when executing the **SQL EXECUTE** command.<br>Values: time in seconds<br>By default: no timeout |
| SQL use access rights | String | SQL_Use_Access_Rights | Used to restrict the access rights to be applied during execution of the SQL commands of the script. When you use this attribute, you must pass 0 or 1 in *attribValue*:<br>• *attribValue* = 1: 4D uses the access rights of the current 4D user.<br>• *attribValue* = 0 (or attribute not specified): 4D does not restrict access, the Designer rights are used. |
| SQL_INTERNAL | String | ;DB4D_SQL_LOCAL; | |
| System data source | Longint | 2 | |
| User data source | Longint | 1 | |

## Standard System Signatures

The Standard System Signatures are 4-character strings designated standard file types, resource types, standard data types stored into the Clipboard and so on.

| Constant | Type | Value | Comment |
|---|---|---|---|
| Picture document | String | PICT | |
| Text document | String | TEXT | |
| Windows MIDI document | String | MID | |
| Windows sound document | String | WAV | |
| Windows video document | String | AVI | |

System Documents

| Constant | Type | Value | Comment |
|---|---|---|---|
| Absolute path | Longint | 2 | The *documents* array contains absolute pathnames |
| Alias selection | Longint | 8 | Authorizes the selection of shortcuts (Windows) or aliases (Mac OS) as document. By default, if this constant is not used, when an alias or shortcut is selected, the command will return the access path of the targeted element. When you pass the constant, the command returns the path of the alias or shortcut itself. |
| Delete only if empty | Longint | 0 | Deletes folder only when it is empty |
| Delete with contents | Longint | 1 | Deletes folder along with everything it contains |
| Document unchanged | Longint | 0 | No processing |
| Document with CR | Longint | 3 | Line breaks are converted to OS X format: CR (carriage return) |
| Document with CRLF | Longint | 2 | Line breaks are converted to Windows format: CRLF (carriage return + line feed) |
| Document with LF | Longint | 4 | Line breaks are converted to Unix format: LF (line feed) |
| Document with native format | Longint | 1 | (Default) Line breaks are converted to the native format of the operating system: CR (carriage return) under OS X, CRLF (carriage return + line feed) under Windows |
| File name entry | Longint | 32 | Allows user to enter a file name in a 'Save as' dialog box. No file is saved and it is up to the developer to create a file in response to this action (the Document system variable is updated). In this context, the *directory* parameter may contain the path to a file instead of a directory. The file name will be used as the suggested file name in the Save as text field. The parent directory will be used as default path. |
| Folder separator | String | Windows="¥" Mac OS=":" | This constant has a different value depending on the operating system from which it is called. It can be used to build valid pathnames without taking the operating platform into account.<br>**Note:** This constant is only used for 4D applications executed in Unicode mode (it is not supported in non-Unicode compatibility mode). |
| Get pathname | Longint | 3 | |
| Ignore invisible | Longint | 8 | Invisible documents are not listed |
| Is a document | Longint | 1 | |
| Is a folder | Longint | 0 | |
| Multiple files | Longint | 1 | Authorizes the simultaneous selection of several files using the key combinations **Shift+click** (adjacent selection) and **Ctrl+click** (Windows) or **Command+click** (Mac OS). In this case, the *selected* parameter, if passed, contains the list of all selected files. By default, if this constant is not used, the command will not allow the selection of multiple files. |
| Package open | Longint | 2 | (Mac OS only): Authorizes the opening of packages as folders and thus the viewing /selection of their contents. By default, if this constant is not used, the command will not allow the opening of packages. |
| Package selection | Longint | 4 | (Mac OS only): Authorizes the selection of packages as entities. By default, if this constant is not used, the command will not allow the selection of software packages as such. |
| Posix path | Longint | 4 | The *documents* array contains Posix format pathnames |
| Read and write | Longint | 0 | Default mode |
| Read mode | Longint | 2 | |
| Recursive parsing | Longint | 1 | The *documents* array contains all files and subfolders of the specified folder |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Use sheet window | Longint | 16 | (Mac OS only): Displays the selection dialog box in the form of a sheet window (this option is ignored under Windows).<br>Sheet windows are specific to the Mac OS X interface which have graphic animation (for more information, refer to the **Window Types** section). By default, if this constant is not used, the command will display a standard dialog box. |
| Write mode | Longint | 1 | |

## System Folder

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_Mac control panels | Longint | 11 | |
| _o_Mac extensions | Longint | 10 | |
| _o_Mac shutdown items_all | Longint | 6 | |
| _o_Mac shutdown items_user | Longint | 7 | |
| Applications or program files | Longint | 16 | |
| Desktop | Longint | 15 | |
| Documents folder | Longint | 17 | "Documents" folder of user |
| Favorites Win | Longint | 14 | |
| Fonts | Longint | 1 | |
| Start menu Win_all | Longint | 8 | |
| Start menu Win_user | Longint | 9 | |
| Startup Win_all | Longint | 4 | |
| Startup Win_user | Longint | 5 | |
| System | Longint | 0 | |
| System Win | Longint | 12 | |
| System32 Win | Longint | 13 | |
| User preferences_all | Longint | 2 | |
| User preferences_user | Longint | 3 | |

## System Format

| Constant | Type | Value | Comment |
|---|---|---|---|
| Currency symbol | Longint | 2 | Currency symbol (e.g.: “$”) |
| Date separator | Longint | 13 | Separator used in date formats (e.g.: “/”) |
| Decimal separator | Longint | 0 | Decimal separator (e.g.: “.”) |
| Short date day position | Longint | 15 | Position of the day in the short date format: “1” = left, “2” = middle, “3” = right |
| Short date month position | Longint | 16 | Position of the month in the short date format: “1” = left, “2” = middle, “3” = right |
| Short date year position | Longint | 17 | Position of the year in the short date format: “1” = left, “2” = middle, “3” = right |
| System date long pattern | Longint | 8 | Long date display format in the form “dddd MMMM yyyy” |
| System date medium pattern | Longint | 7 | Medium date display format in the form “dddd MMMM yyyy” |
| System date short pattern | Longint | 6 | Short date display format in the form “dddd MMMM yyyy” |
| System time AM label | Longint | 18 | Additional label for a time before noon in 12-hour formats (e.g.: “Morning”) |
| System time long pattern | Longint | 5 | Long time display format in the form “HH:MM:SS” |
| System time medium pattern | Longint | 4 | Medium time display format in the form “HH:MM:SS” |
| System time PM label | Longint | 19 | Additional label for a time after noon in 12-hour formats (e.g.: “Afternoon”) |
| System time short pattern | Longint | 3 | Short time display format in the form “HH:MM:SS” |
| Thousand separator | Longint | 1 | Thousand separator (e.g.: “,”) |
| Time separator | Longint | 14 | Separator used in time formats (e.g.: “:”) |

## TCP Port Numbers

| Constant | Type | Value | Comment |
|---|---|---|---|
| TCP Authentication | Longint | 113 | |
| TCP DNS | Longint | 53 | |
| TCP Finger | Longint | 79 | |
| TCP FTP Control | Longint | 21 | |
| TCP FTP Data | Longint | 20 | |
| TCP Gopher | Longint | 70 | |
| TCP HTTP WWW | Longint | 80 | |
| TCP IMAP3 | Longint | 220 | |
| TCP Kerberos | Longint | 88 | |
| TCP KLogin | Longint | 543 | |
| TCP Nickname | Longint | 43 | |
| TCP NNTP | Longint | 119 | |
| TCP NTalk | Longint | 518 | |
| TCP NTP | Longint | 123 | |
| TCP PMCP | Longint | 1643 | |
| TCP PMD | Longint | 1642 | |
| TCP POP3 | Longint | 110 | |
| TCP Printer | Longint | 515 | |
| TCP RADACCT | Longint | 1646 | |
| TCP RADIUS | Longint | 1645 | |
| TCP Remote Cmd | Longint | 514 | |
| TCP Remote Exec | Longint | 512 | |
| TCP Remote Login | Longint | 513 | |
| TCP Router | Longint | 520 | |
| TCP SMTP | Longint | 25 | |
| TCP SNMP | Longint | 161 | |
| TCP SNMPTRAP | Longint | 162 | |
| TCP SUN RPC | Longint | 111 | |
| TCP Talk | Longint | 517 | |
| TCP Telnet | Longint | 23 | |
| TCP TFTP | Longint | 69 | |
| TCP UUCP | Longint | 540 | |
| TCP UUCP RLOGIN | Longint | 541 | |

## Text Values for Associated Standard Action

| Constant | Type | Value | Comment |
|---|---|---|---|
| Object Accept action | String | 2 | |
| Object Add subrecord action | String | 14 | |
| Object Automatic splitter action | String | 16 | |
| Object Cancel action | String | 1 | |
| Object Clear action | String | 21 | |
| Object Copy action | String | 19 | |
| Object Cut action | String | 18 | |
| Object Database Settings action | String | 32 | |
| Object Delete record action | String | 7 | |
| Object Delete subrecord action | String | 13 | |
| Object Edit subrecord action | String | 12 | |
| Object First page action | String | 10 | |
| Object First record action | String | 5 | |
| Object Goto page action | String | 15 | |
| Object Last page action | String | 11 | |
| Object Last record action | String | 6 | |
| Object MSC action | String | 36 | |
| Object Next page action | String | 8 | |
| Object Next record action | String | 3 | |
| Object No standard action | String | 0 | |
| Object Open back URL action | String | 37 | |
| Object Open next URL action | String | 38 | |
| Object Paste action | String | 20 | |
| Object Previous page action | String | 9 | |
| Object Previous record action | String | 4 | |
| Object Quit action | String | 27 | |
| Object Redo action | String | 31 | |
| Object Refresh current URL action | String | 39 | |
| Object Return to Design mode action | String | 35 | |
| Object Select all action | String | 22 | |
| Object Show Clipboard action | String | 23 | |
| Object Stop loading URL action | String | 40 | |
| Object Test Application action | String | 26 | |
| Object Undo action | String | 17 | |

## Time Display Formats

| Constant | Type | Value | Comment |
|---|---|---|---|
| Blank if null time | Longint | 100 | "" instead of 0 |
| HH MM | Longint | 2 | 01:02 |
| HH MM AM PM | Longint | 5 | 1:02 AM |
| HH MM SS | Longint | 1 | 01:02:03 |
| Hour min | Longint | 4 | 1 hour 2 minutes |
| Hour min sec | Longint | 3 | 1 hour 2 minutes 3 seconds |
| ISO time | Longint | 8 | 0000-00-00T01:02:03 |
| Min sec | Longint | 7 | 62 minutes 3 seconds |
| MM SS | Longint | 6 | 62:03 |
| System time long | Longint | 11 | 1:02:03 AM HNEC (Mac only) |
| System time long abbreviated | Longint | 10 | 1•02•03 AM (Mac only) |
| System time short | Longint | 9 | 01:02:03 |

## Trigger Events

| Constant | Type | Value | Comment |
|---|---|---|---|
| _o_On Loading Record Event | Longint | 4 | |
| On Deleting Record Event | Longint | 3 | |
| On Saving Existing Record Event | Longint | 2 | |
| On Saving New Record Event | Longint | 1 | |

## Value for Associated Standard Action

All constants in this theme are deprecated starting with 4D v16 R3. You should use text constants from the **Text Values for Associated Standard Action** theme.

| Constant | Type | Value | Comment |
|---|---|---|---|
| Accept action | Longint | 2 | |
| Add subrecord action | Longint | 14 | |
| Cancel action | Longint | 1 | |
| Clear action | Longint | 21 | |
| Copy action | Longint | 19 | |
| Cut action | Longint | 18 | |
| Database settings action | Longint | 32 | |
| Delete record action | Longint | 7 | |
| Delete subrecord action | Longint | 13 | |
| Edit subrecord action | Longint | 12 | |
| First page action | Longint | 10 | |
| First record action | Longint | 5 | |
| Last page action | Longint | 11 | |
| Last record action | Longint | 6 | |
| MSC action | Longint | 36 | |
| Next page action | Longint | 8 | |
| Next record action | Longint | 3 | |
| No action | Longint | 0 | |
| Paste action | Longint | 20 | |
| Previous page action | Longint | 9 | |
| Previous record action | Longint | 4 | |
| Quit action | Longint | 27 | |
| Redo action | Longint | 31 | |
| Return to Design mode | Longint | 35 | |
| Select all action | Longint | 22 | |
| Show clipboard action | Longint | 23 | |
| Test application action | Longint | 26 | |
| Undo action | Longint | 17 | |

## Web Area

| Constant | Type | Value | Comment |
|---|---|---|---|
| WA enable contextual menu | Longint | 4 | Allow the display of a standard contextual menu in the Web area |
| WA enable Java applets | Longint | 1 | Allow the execution of Java applets in the Web area. |
| WA enable JavaScript | Longint | 2 | Allow the execution of JavaScript code in the Web area |
| WA enable plugins | Longint | 3 | Allow the installation of plug-ins in the Web area |
| WA enable URL drop | Longint | 101 | Allow dropping of URLs or files in the Web area (default = False) |
| WA enable Web inspector | Longint | 100 | Allow the display of the Web inspector in the area |
| WA next URLs | Longint | 1 | |
| WA previous URLs | Longint | 0 | |

## Web Server

| Constant | Type | Value | Comment |
|---|---|---|---|
| wdl disable | Longint | 0 | Web HTTP debug log is disabled |
| wdl enable with all body parts | Longint | 7 | Web HTTP debug log is enabled with body parts in response and request |
| wdl enable with request body | Longint | 5 | Web HTTP debug log is enabled with body part in request only |
| wdl enable with response body | Longint | 3 | Web HTTP debug log is enabled with body part in response only |
| wdl enable without body | Longint | 1 | Web HTTP debug log is enabled without body parts (body size is provided in this case) |
| Web character set | Longint | 17 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Character set that the 4D Web Server (with 4D in local mode and 4D Server) should use to communicate with browsers connecting to the database. The default value actually depends on the language of the operating system. This parameter is set in the Database settings.<br>**Possible values**: The possible values depend on the operating mode of the database relating to the character set.<br>• *Unicode Mode*: When the application is operating in Unicode mode, the values to pass for this parameter are character set identifiers (*MIBEnum* longint or *Name* string, identifiers defined by IANA, see the following address: http://www.iana.org/assignments/character-sets). Here is the list of identifiers corresponding to the character sets supported by the 4D Web server:<br>4=ISO-8859-1<br>12=ISO-8859-9<br>13=ISO-8859-10<br>17=Shift-JIS<br>2024=Windows-31J<br>2026=Big5<br>38=euc-kr<br>106=UTF-8<br>2250=Windows-1250<br>2251=Windows-1251<br>2253=Windows-1253<br>2255=Windows-1255<br>2256=Windows-1256<br>• *ASCII compatibility mode*:<br>Western European<br>1: Japanese<br>2: Chinese<br>3: Korean<br>4: User-defined<br>5: Reserved<br>6: Central European<br>7: Cyrillic<br>8: Arabic<br>9: Greek<br>10: Hebrew<br>11: Turkish<br>12: Baltic |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Web debug log | Longint | 84 | **Scope:** Local Web server<br>**Kept between two sessions:** No, but remains valid even if the HTTP server is restarted (a new log file is used in this case)<br>**Description:** Allows you to get or set the status of the HTTP request log file of the 4D Web server. When enabled, this file, named "**HTTPDebugLog_*nn*.txt**", is stored in the "Logs" folder of the application (*nn* is the file number). It is useful for debugging issues related to the Web server. It records each request and each response in raw mode. Whole requests, including headers, are logged; optionally, body parts can be logged as well.<br>**Values**: One of the constants prefixed with "wdl" (refer to the descriptions of these constants in this theme).<br>**Default value**: 0 (not enabled) |
| Web HTTP compression level | Longint | 50 | **Scope**: Local Web server<br>**Kept between two sessions**: No<br>**Description**: Compression level for all compressed HTTP exchanges for the 4D HTTP server (client requests or server replies, Web and Web Service). This selector lets you optimize exchanges by either privileging speed of execution (less compression) or the amount of compression (less speed). The choice of a value depends on the size and type of data exchanged. Pass 1 to 9 in the *value* parameter where 1 is the fastest compression and 9 the highest. You can also pass -1 to get a compromise between speed and rate of compression. By default, the compression level is 1 (faster compression).<br>**Possible values**: 1 to 9 (1 = faster, 9 = more compressed) or -1 = best compromise. |
| Web HTTP compression threshold | Longint | 51 | **Scope**: Local HTTP server<br>**Kept between two sessions**: No<br>**Description**: In the framework of optimized HTTP exchanges, size threshold for requests below which exchanges should not be compressed. This setting is useful in order to avoid losing machine time by compressing small exchanges.<br>**Possible values**: Any Longint type value. Pass the size expressed in bytes in *vaue*. By default, the compression threshold is set to 1024 bytes |
| Web HTTP TRACE | Longint | 85 | **Scope**: Local Web server<br>**Kept between two sessions**: No<br>**Description**: Allows you to disable or enable the HTTP TRACE method in the 4D Web server. For security reasons, starting with 4D v15 R2, by default the 4D Web server rejects HTTP TRACE requests with an error 405 (see HTTP TRACE disabled). If necessary, you can enable the HTTP TRACE method for the session by passing this constant with value 1. When this option is enabled, the 4D Web server replies to HTTP TRACE requests with the request line, header, and body.<br>**Possible values**: 0 (disabled) or 1 (enabled)<br>**Default value**: 0 (disabled) |
| Web HTTPS port ID | Longint | 39 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: TCP port number used by the Web server of 4D in local mode and of 4D Server for secure connections via SSL (HTTPS protocol). The HTTPS port number is set on the “Web/Configuration” page of the Database settings dialog box.<br>By default, the value is 443 (standard value). You can use the constants of the **TCP Port Numbers** theme for the *value* parameter.<br>**Possible values**: 0 to 65535 |
| Web inactive process timeout | Longint | 78 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description:** Modifies the life duration of the inactive processes associated with sessions. At the end of the timeout, the process is killed on the server, the **On Web Close Process database method** is called then the session context is destroyed.<br>**Possible values**: Longint (minutes)<br>**Default value**: 480 minutes (pass 0 to restore the default value) |
| Web inactive session timeout | Longint | 72 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description**: Modifies the life duration of inactive sessions (duration set in cookie). At the end of this period, the session cookie expires and is no longer sent by the HTTP client.<br>**Possible values**: Longint (minutes)<br>**Default value**: 480 minutes (pass 0 to restore the default value) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Web IP address to listen | Longint | 16 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: IP address on which the 4D Web server will receive HTTP requests with 4D in local mode and 4D Server. By default, no specific address is defined (*value* = 0). This parameter can be set in the Database settings. The *Web IP Address to listen* selector is useful for 4D Web Servers compiled and merged with 4D Desktop (in which there is no access to the Design mode).<br>You pass a hexadecimal IP address in the *value* parameter. In other words, to designate a IP address such as "a.b.c.d", you should write:<br>`C_LONGINT($addr)`<br>`$addr:=($a<<24)\|($b<<16)\|($c<<8)\|$d`<br>`WEB SET OPTION(Web IP address to listen;$addr)` |
| Web keep session | Longint | 70 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description:** Enables or disables the session management mode (described in the **Web Sessions Management** section)<br>**Possible values**: 1 (enable mode) or 0 (disable mode)<br>**Default value**: 1 for databases created in v13, 0 for converted databases. Note that this mode also enables the mechanism for reusing temporary contexts in remote mode. For more information about this mechanism, refer to the description of this option in the **Web Server Settings** section. |
| Web log recording | Longint | 29 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Starts or stops the recording of Web requests received by the Web server of 4D in local mode or 4D Server. By default, the value is 0 (requests not recorded).<br>The log of Web requests is stored as a text file named "logweb.txt" that is automatically placed in the Logs folder of the database, next to the structure file. The format of this file is determined by the value that you pass. For more information about Web log file formats, please refer to the **Information about the Web Site** section.<br>This file can also be activated on the "Web/Log" page of the Database settings.<br>**Possible values**: 0 = Do not record (default), 1 = Record in CLF format, 2 = Record in DLF format, 3 = Record in ELF format, 4 = Record in WLF format.<br><br>**Warning**: Formats 3 and 4 are custom formats whose contents must be set beforehand in the Database settings. If you use one of these formats without any of its fields having been selected on this page, the log file will not be generated. |
| Web max concurrent processes | Longint | 18 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Strictly upper limit of concurrent Web processes of any type supported by the 4D Web Server with 4D in local mode and 4D Server. When this number (minus one) is reached, 4D will not create any other processes and returns the HTTP status 503 - Service Unavailable to all new requests.<br>This parameter can prevent the 4D Web Server from saturation, which can occur when an exceedingly large number of concurrent requests are sent, or when too many context creations are requested. This parameter can also be set in the Database settings.<br>In theory, the maximum number of Web processes is the result of the following formula: Available memory/Web process stack size. Another solution is to view the information on Web processes displayed in the Runtime Explorer: the current number of Web processes and the maximum number reached since the Web server boot are indicated.<br>**Possible values**: Any value between 10 and 32 000. The default value is 100. |
| Web max sessions | Longint | 71 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description**: Limits the number of simultaneous sessions. When you reach the limit set, the oldest session is closed (and **On Web Close Process database method** is called) if the Web server needs to create a new one.<br>**Possible values**: Longint. The number of simultaneous sessions cannot exceed the total number of Web processes (Web Max Concurrent Processes option, 100 by default)<br>**Default value**: 100 (pass 0 to restore the default value) |

| Constant | Type | Value | Comment |
|---|---|---|---|
| Web maximum requests size | Longint | 27 | **Scope**: 4D local, 4D Server<br>**Kept between two sessions**: Yes<br>**Description**: Maximum size (in bytes) of incoming HTTP requests (POST) that the Web server is authorized to process. By default, the value is 2 000 000, i.e. a little less than 2 MB. Passing the maximum value (2 147 483 648) means that, in practice, no limit is set.<br>This limit is used to avoid Web server saturation due to incoming requests that are too large. When a request reaches this limit, the 4D Web server refuses it.<br>**Possible values**: 500 000 to 2 147 483 648. |
| Web port ID | Longint | 15 | **Scope**: 4D in local mode and 4D Server.<br>**Kept between two sessions:** No<br>**Description:** Sets or gets the number of the TCP port used by the 4D Web server with 4D in local mode and 4D Server. By default, the value is 80. The TCP port number is set on the "Web/Configuration" page of the Database Settings dialog box. You can use one of the constants in the **TCP Port Numbers** theme for the *value* parameter. This selector is useful within the framework of 4D Web servers that are compiled and merged using 4D Desktop (no access to the Design environment).<br>**Possible values:** For more information about the TCP port number, refer to the **Web Server Settings** section.<br>**Default value:** 80 |
| Web session cookie domain | Longint | 81 | **Scope:** local Web server<br>**Kept between two sessions:** No, but remains valid even if the HTTP server is restarted.<br>**Description:** Sets or gets the value of the "domain" field of the session cookie. This selector (as well as selector 82) is useful for controlling the scope of the session cookies: If you set, for example, the value "/*.4d.fr" for this selector, the client will only send a cookie when the request is addressed to the domain ".4d.fr", which excludes servers hosting external static data.<br>**Possible values:** Text |
| Web session cookie name | Longint | 73 | **Scope**: Local Web server<br>**Kept between two sessions**: No, but remains valid even if the HTTP server is restarted<br>**Description:** Sets the name of the cookie used for saving the session ID.<br>**Possible values:** Text<br>**Default value:** "4DSID" (pass an empty string to restore the default value) |
| Web session cookie path | Longint | 82 | **Scope:** local Web server<br>**Kept between two sessions:** No, but remains valid even if the HTTP server is restarted.<br>**Description:** Sets or gets the value of the "path" field of the session cookie. This selector (as well as selector 81) is useful for controlling the scope of the session cookies: If you set, for example, the value "/4DACTION" for this selector, the client will only send a cookie for dynamic requests beginning with 4DACTION, and not for pictures, static pages, etc.<br>**Possible values:** Text |
| Web session enable IP address validation | Longint | 83 | **Scope**: Local Web server<br>**Kept between two sessions**: No<br>**Description**: Enables or disables IP address validation for session cookies. For security reasons, by default the 4D Web server checks the IP address of each request containing a session cookie and rejects it if this address does not match the IP address used to create the cookie. In some specific applications, you may want to disable this validation and accept session cookies, even when their IP addresses do not match. For example when mobile devices switch between Wifi and 3G/4G networks, their IP address will change. In this case, you must pass 0 in this option to allow clients to be able to continue using their Web sessions even when the IP addresses change. Note that this setting lowers the security level of your application.<br>When it is modified, this setting is effective immediately (you do not need to restart the HTTP server).<br>**Possible values**: 0 (disabled) or 1 (enabled)<br>**Default value**: 1 (IP addresses are checked) |

Web Services (Client)

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Web Service compression | Longint | 1 | |
| Web Service detailed message | Longint | 1 | Detailed message describing the error. The type of message differs according to the main error type.<br>- If the main error = 9910 (Soap fault): the cause of the SOAP fault is returned (e.g.: “the remote method does not exist”).<br>- If the main error = 9911 (Parser fault): the location of the error in the XML document is returned.<br>- If the main error = 9912 (HTTP fault):<br>- if the HTTP error is located in the interval [300-400] (problems linked to the location of the requested document), the new location of the requested URL is returned.<br>- for any other HTTP error code, the <body> is returned.<br>- If the main error = 9913 (Network fault): the cause of the network fault is returned (e.g.: “ServerAddress: DNS lookup failure”)<br>- If the main error = 9914 (Internal fault): the cause of the internal fault is returned |
| Web Service display auth dialog | Longint | 4 | *value* = 0 (do not display dialog box) or 1 (display dialog box)<br>This option manages the display of the authentication dialog box during execution of the **WEB SERVICE CALL** command. By default, this command never displays the dialog box; normally, you have to use the **WEB SERVICE AUTHENTICATE** command to do so. However, if you want the authentication dialog box to appear in order for the user to enter their identifiers, you will need to use this option: pass 1 in *value* to display the dialog box and 0 otherwise. The dialog box only appears if the Web service requires authentication |
| Web Service dynamic | Longint | 0 | |
| Web Service error code | Longint | 0 | Main error code (defined by 4D). This code is also returned in the Error system variable.<br>List of codes that may be returned:<br>9910: Soap fault (see also Web Service Fault Actor)<br>9911: Parser fault<br>9912: HTTP fault (see also Web Service HTTP Error code)<br>9913: Network fault<br>9914: Internal fault. |
| Web Service fault actor | Longint | 3 | Cause of the error (returned by the SOAP protocol — to be used in the case of main error 9910).<br>- Version Mismatch<br>- Must Understand (the server was unable to interpret a parameter defined as mandatory)<br>- Sender Fault<br>- Receiver Fault<br>- Encoding Unknown |
| Web Service HTTP compression | Longint | 6 | *value* = Web Service Compression<br>This option is used to enable an internal compression mechanism for SOAP requests in order to accelerate inter-4D application exchanges. When you execute the statement **WEB SERVICE SET OPTION**(Web Service HTTP Compression; Web Service Compression) on the 4D client of the Web Service, the data of the next SOAP request sent by the client will be compressed using a standard HTTP mechanism ("gzip" or "deflate", depending on the request contents) before being sent to the 4D SOAP server. The server will decompress and parse the request, then will reply automatically using the same mechanism. Only the request that follows the call to the **WEB SERVICE SET OPTION** command is affected. You must therefore call this command each time you want to use compression. By default, 4D does not compress Web Service HTTP requests.<br>**Note:** This mechanism cannot be used for requests sent to a 4D SOAP server whose version is earlier than 11.3. So that you can further optimize this functioning, additional options configure the threshold and compression rate of the requests. These options can be accessed via the **SET DATABASE PARAMETER** command |
| Web Service HTTP error code | Longint | 2 | HTTP error code (to be used in case of main error 9912). |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| Web Service HTTP timeout | Longint | 1 | *value* = timeout of the client portion expressed in seconds.<br>The timeout of the client portion is the wait period of the Web Service client in case the server does not respond. After this period, the client closes the session and the request is lost.<br>This timeout is 180 seconds by default. It can be modified for specific reasons (network status, Web Service specifics, etc.). |
| Web Service manual | Longint | 3 | |
| Web Service manual in | Longint | 1 | |
| Web Service manual out | Longint | 2 | |
| Web Service reset auth settings | Longint | 5 | *value* = 0 (do not erase information) or 1 (erase information)<br>This option lets you indicate to 4D whether to memorize the authentication information of the user (user name, password, method, etc.), in order to reuse it subsequently. By default, this information is erased after each execution of the **WEB SERVICE CALL** command. Pass 0 in value to store the information and 1 to erase it. Note that when you pass 0, the information is kept during the session but is not stored. |
| Web Service SOAP header | Longint | 2 | *value* = XML root element reference to insert as a header in the SOAP request.<br>This option allows you to insert a header in a SOAP request generated using the **WEB SERVICE CALL** command. SOAP requests do not contain a specific header by default. However, certain Web Services require a header, for example when managing identification parameters |
| Web Service SOAP version | Longint | 3 | *value* = Web Service SOAP_1_1 or Web Service SOAP_1_2<br>This option lets you specify the SOAP protocol version used in the request. Pass the Web Service SOAP_1_1 constant in value to indicate version 1.1 and Web Service SOAP_1_2 to indicate version 1.2. |
| Web Service SOAP_1_1 | Longint | 0 | |
| Web Service SOAP_1_2 | Longint | 1 | |

## Web Services (Server)

| Constant | Type | Value | Comment |
|---|---|---|---|
| Is DOM reference | Longint | 37 | |
| Is XML | Longint | 36 | |
| SOAP client fault | Longint | 1 | |
| SOAP input | Longint | 1 | |
| SOAP method name | Longint | 1 | Name of the Web Service method about to be executed |
| SOAP output | Longint | 2 | |
| SOAP server fault | Longint | 2 | |
| SOAP service name | Longint | 2 | Name of the Web Service to which the method belongs |

## Windows

| Constant | Type | Value | Comment |
|---|---|---|---|
| External window | Longint | 5 | |
| Floating window | Longint | 14 | |
| Modal dialog | Longint | 9 | |
| Regular window | Longint | 8 | |
| XY Current form | Longint | 1 | Origin is top left corner of current form |
| XY Current window | Longint | 2 | Origin is top left corner of current window |
| XY Main window | Longint | 4 | On Windows: origin is top left corner of main window; on OS X: same as XY Screen |
| XY Screen | Longint | 3 | Origin is top left corner of main screen (same as for **SCREEN COORDINATES** command) |

XML

| Constant | Type | Value | Comment |
|---|---|---|---|
| Copy XML data source | Longint | 1 | 4D keeps a copy of the DOM tree with the picture, which means the picture can be saved in a picture field of the database and then redisplayed or exported at any time. |
| DOCTYPE Name | Longint | 3 | Name of the root element as defined in the DOCTYPE marker |
| Document URI | Longint | 6 | URI of the DTD |
| Encoding | Longint | 4 | Encoding used (UTF-8, ISO...) |
| Get XML data source | Longint | 0 | 4D only reads the XML data source; it is not kept with the picture. This noticeably increases command execution speed; however, because the DOM tree is not kept, it is not possible to store or export the picture. |
| Own XML data source | Longint | 2 | 4D exports the DOM tree with the picture. The picture can be stored or exported and command execution is fast. However, the *elementRef* XML reference can then no longer be used by other 4D commands. This is the default mode for exporting when the *exportType* parameter is omitted. |
| PUBLIC ID | Longint | 1 | Public identifier (FPI) of the DTD to which the document conforms (if the DOCTYPE xxx PUBLIC tag is present). |
| SYSTEM ID | Longint | 2 | System identifier |
| Version | Longint | 5 | Accepted XML version |
| XML Base64 | Longint | 1 | |
| XML binary encoding | Longint | 5 | Specifies the way binary data will be converted.<br>**Possible values:**<br>• XML Base64 (default value): binary data are simply converted to Base64<br>• XML data URI scheme: binary data are converted to Base64 and the "data:;base64" header is added. This format mainly allows a browser to automatically decode a picture, and is also required for the insertion of pictures. For more information, see http://en.wikipedia.org/wiki/Data_URI_scheme. |
| XML CDATA | Longint | 7 | |
| XML comment | Longint | 2 | |
| XML convert to PNG | Longint | 1 | |
| XML DATA | Longint | 6 | |
| XML data URI scheme | Longint | 2 | |
| XML date encoding | Longint | 2 | Specifies the way 4D dates will be converted. For example, !01/01/2003! in the Paris time zone.<br>**Possible values:**<br>• XML ISO (default value): use of the format xs:datetime without indication of time zone. Result: "2003-01-01". The time part, if it is present in the 4D value (via SQL) is lost.<br>• XML local: use of the format xs:date with indication of time zone. Result: "2003-01-01 +01:00". The time part, if it is present in the 4D value (via SQL) is lost.<br>• XML datetime local: use of the format xs:dateTime (ISO 8601). Indication of time zone. This format allows the time part to be kept, if it is present in the 4D value (via SQL). Result: "<Date>2003-01-01T00:00:00 +01:00</Date>".<br>• XML UTC: use of the format xs:date. Result: "2003-01-01Z". The time part, if it is present in the 4D value (via SQL) is lost.<br>• XML datetime UTC: use of the format xs:dateTime (ISO 8601). This format allows the time part to be kept, if it is present in the 4D value (via SQL). Result: "<Date>2003-01-01T00:00:00Z</Date>". |

| Constant | Type | Value | Comment |
|---|---|---|---|
| XML datetime local | Longint | 3 | |
| XML datetime local absolute | Longint | 1 | |
| XML datetime UTC | Longint | 5 | |
| XML DOCTYPE | Longint | 10 | |
| XML duration | Longint | 2 | |
| XML ELEMENT | Longint | 11 | |
| XML end document | Longint | 9 | |
| XML end element | Longint | 5 | |
| XML entity | Longint | 8 | |
| XML indentation | Longint | 4 | Specifies the indentation of the XML *document*.<br>**Possible values:**<br>• XML with indentation (default value): the document is indented.<br>• XML no indentation: the document is not indented; its contents are placed in a single line. |
| XML ISO | Longint | 1 | |
| XML local | Longint | 2 | |
| XML native codec | Longint | 2 | |
| XML no indentation | Longint | 2 | |
| XML picture encoding | Longint | 6 | Specifies the way pictures must be converted (before encoding in Base64).<br>**Possible values:**<br>• XML convert to PNG (default value): pictures are converted to PNG before being encoded in Base64.<br>• XML native codec: pictures are converted in their first native storage CODEC before being encoded in Base64. You must use these options to encode SVG pictures (see example for the **XML SET OPTIONS** command). |
| XML processing instruction | Longint | 3 | |
| XML raw data | Longint | 2 | |
| XML seconds | Longint | 4 | |
| XML start document | Longint | 1 | |
| XML start element | Longint | 4 | |

| Constant | Type | Value | Comment |
| --- | --- | --- | --- |
| XML string encoding | Longint | 1 | Specifies the way 4D strings are converted to element values. It does not concern the conversion to attributes for which XML imposes the use of escape characters.<br>**Possible values:**<br>• XML with escaping (default value): conversion of 4D strings to XML element values with replacement of characters. The Text type data are automatically parsed so that forbidden characters (<&>') are replaced by XML entities (&<> '").<br>• XML raw data: 4D strings are sent as raw data; 4D does not carry out encoding or parsing. 4D values are converted if possible to XML fragments and inserted as a child of the target element. If a value cannot be considered as an XML fragment, it is inserted as raw data into a new CDATA node. |
| XML time encoding | Longint | 3 | Specifies the way 4D times are converted. For example, ?02/00/46? (Paris time). The encoding differs depending on whether you want to express a time or a duration.<br>**Possible values for times:**<br>• XML datetime UTC: time expressed in UTC (Universal Time Coordinated). Note that conversion to UTC is automatic. Result: "<Duration>0000-00-00T01:00:46Z</Duration>".<br>• XML datetime local: time expressed with the time difference of the machine of the 4D engine. Result: "<Duration>0000-00-00T02:00:46+01:00</Duration>".<br>• XML datetime local absolute (default value): time expressed without indication of time zone. No modification of the value. Result: "<Duration>0000-00-00T02:00:46</Duration>".<br>**Possible values for durations:**<br>• XML seconds: number of seconds since midnight; no modification of the value since it expresses a duration. Result: "<Duration>7246</Duration>".<br>• XML duration: duration expressed in compliance with XML Schema Part 2: Datatypes Second Edition. No modification of the value since it expresses a duration. Result: "<Duration>PT02H00M46S</Duration>". |
| XML UTC | Longint | 4 | |
| XML with escaping | Longint | 1 | |
| XML with indentation | Longint | 1 | |

# Error Codes

- Syntax Errors (1 -> 81)
- Database Engine Errors (-10602 -> 4004)
- SQL Engine Errors (1001 -> 3018)
- Network Errors (-10051 -> -10001)
- Backup Manager Errors (1401 -> 1421)
- Client Update Errors (-10650 -> -10655)
- Updater Errors (-10603 -> -10613)
- OS File Manager Errors (-124 -> -33)
- OS Memory Manager Errors (-117 -> -108)
- OS Printing Manager Errors (-8192 -> -1)
- OS Resource Manager Errors (-196 -> -1)
- SANE NaN Errors (1 -> 255)
- OS Sound Manager Errors (-209 -> -203)
- OS Serial Ports Manager Errors (-28)
- Mac OS System Errors (4 -> 28)
- Miscellaneous Errors (-10700)

## Syntax Errors (1 -> 81)

---

The following table lists the syntax error codes for errors that may occur during code execution in the Design or Application environment. Some of these errors may occur in interpreted mode only, some in compiled mode only, some in both modes. You can intercept these errors using an error interruption method installed using **ON ERR CALL**.

| Code | Description |
|---|---|
| 1 | A “(” was expected. |
| 2 | A field was expected. |
| 3 | The command may be executed only on a field in a subtable. |
| 4 | Parameters in the list must all be of the same type. |
| 5 | There is no table to which to apply the command. |
| 6 | The command may only be executed on a Subtable type field. |
| 7 | A Numeric argument was expected. |
| 8 | An Alphanumeric argument was expected. |
| 9 | The result of a conditional test was expected. |
| 10 | The command cannot be applied to this field type. |
| 11 | The command cannot be applied between two conditional tests. |
| 12 | The command cannot be applied between two Numeric arguments. |
| 13 | The command cannot be applied between two Alphanumeric arguments. |
| 14 | The command cannot be applied between two Date arguments. |
| 15 | The operation is not compatible with the two arguments. |
| 16 | The field has no relation. |
| 17 | A table was expected. |
| 18 | Field types are incompatible. |
| 19 | The field is not indexed. |
| 20 | An “=” was expected. |
| 21 | The method does not exist. |
| 22 | The fields must belong to the same table or subtable for a sort or graph. |
| 23 | A “<” or “>” was expected. |
| 24 | A “;” was expected. |
| 25 | There are too many fields for a sort. |
| 26 | The field type cannot be Text, Picture, Blob or Subtable. |
| 27 | The field must be prefixed by the name of its table. |
| 28 | The field type must be Numeric. |
| 29 | The value must be 1 or 0. |
| 30 | A variable was expected. |
| 31 | There is no menu bar with this number. |
| 32 | A date was expected. |
| 33 | Unimplemented command or function. |
| 34 | Accounting files are not open. |
| 35 | The sets are from different tables. |
| 36 | Invalid table name. |
| 37 | A “:=” was expected. |
| 38 | This is a function, not a procedure. |
| 39 | The set does not exist. |
| 40 | This is a procedure, not a function. |
| 41 | A variable or field belonging to a subtable was expected. |
| 42 | The record cannot be pushed onto the stack. |
| 43 | The function cannot be found. |
| 44 | The method cannot be found. |
| 45 | Field or variable expected. |
| 46 | A Numeric or Alphanumeric argument was expected. |
| 47 | The field type must be Alphanumeric. |
| 48 | Syntax error. |
| 49 | This operator cannot be used here. |
| 50 | These operators cannot be used together. |
| 51 | Module not implemented. |

| | |
|---|---|
| 52 | An array was expected. |
| 53 | Indice out of range. |
| 54 | Argument types are incompatible. |
| 55 | A Boolean argument was expected. |
| 56 | Field, variable, or table expected. |
| 57 | An operator was expected. |
| 58 | A “)” was expected. |
| 59 | This kind of argument was not expected here. |
| 60 | A parameter or a local variable cannot be used in an *EXECUTE* statement in a compiled database. |
| 61 | The type of an array cannot be modified in a compiled database. |
| 62 | The command cannot be applied to a subtable. |
| 63 | The field is not indexed. |
| 64 | A picture field or variable was expected. |
| 65 | The value should contain 4 characters. |
| 66 | The value should not contain more than 3 characters. |
| 67 | This command cannot be executed on 4D Server. |
| 68 | A list was expected. |
| 69 | An external window reference was expected. |
| 70 | The command cannot be applied between two Picture arguments. |
| 71 | The **SET PRINT MARKER** command can only be called in the header of a form being printed. |
| 72 | A pointer array was expected. |
| 73 | A numeric array was expected. |
| 74 | The size of arrays does not match. |
| 75 | No pointer on local arrays. |
| 76 | Bad array type. |
| 77 | Bad variable name. |
| 78 | Invalid sort parameter. |
| 79 | This command cannot be executed during the draw of a list. |
| 80 | Too many query arguments. |
| 81 | The form was not found. |

## Tips

Some of these error codes denote plain syntax errors due to mistyping. For example, you get an error *#37* if you execute the statement *v=0* when you actually meant *v:=0*. You can eliminate the error by fixing your code in the Method editor.

Some of these error codes are due to simple programming errors. For example, you get an error *#5* if you execute an **ADD RECORD** command, when you have not first set the default table (using the **DEFAULT TABLE** command), and do not pass the table parameter. In this case, there is no table to which to apply the command. You eliminate the error by checking to see if you forgot to set the default table or if you forgot to pass the table parameter to the command for this occurrence.

Some of these error codes denote errors due to a flaw in the design. For example, you get an error *#16* if you apply **RELATE ONE** to a field that is not related to any other field. You eliminate the error by checking to see if your code is actually wrong or if you simply forgot to create the relation starting from the field.

Some errors, when they occur, are not always located exactly where your code breaks. For example, if in a subroutine you get an error *#53* (indice out of range) on the line *vpFld:=Field($1;$2)*, the error is due to a wrong table and/or field number that has been passed to the subroutine as a parameter. Therefore, the error is located in the caller method and not where the error actually occurs. In this case, trace your code in the Debugger window to determine which line of code is the real culprit, then fix it in the Method editor.

## Database Engine Errors (-10602 -> 4004)

---

This table lists the error codes generated by the 4D Database Engine. These codes cover errors that occur at a low level of the database engine, such as user interruption, privilege errors, and damaged objects.

| Code | Description |
|---|---|
| -10602 | Can't execute the given command because there is no open print job. |
| -10601 | The **OBJECT DUPLICATE** command does not work during printing operations. |
| -10600 | This BLOB could not be read. It may be damaged. |
| -10532 | Cannot call error handling project method 'methodName. |
| -10526 | Can't create database {database_name}. |
| -10525 | Can't delete existing database {database_name}. |
| -10524 | Can't open database {database_name}: current data file not found at last known location: '{datafile_path}'. |
| -10523 | Can't open database {database_name}: current data file is not defined. |
| -10522 | Can't load component {database_name} because it's not in Unicode mode. |
| -10521 | Can't open database {database_name} because it's not in Unicode mode. |
| -10520 | Unauthorized user: Only an Administrator or Designer can execute this command. |
| -10519 | Bad url: {url} |
| -10518 | Assert failed: {assertion} |
| -10517 | Failed to synchronize {folder_name} folder. |
| -10516 | The server is running a maintenance operation, please reconnect later. |
| -10515 | Your attempt to connect to 4D Server has been denied |
| -10514 | The maximum number of concurrent users for your license has been reached |
| -10513 | Can't call {command_name} command from a remote 4D Developer. |
| -10512 | The encoding is not supported |
| -10511 | Can't call command "{command_name}" from a component. |
| -10510 | Can't load component "{component_name}". |
| -10509 | Can't open database "{database_name}". |
| -10508 | Project method not found. |
| -10507 | This version does not allow a compiled database to be opened. |
| -10506 | Limit of the Standard Edition version. |
| -10505 | Client and server have version numbers that are incompatible. |
| -10504 | Index page # is out of range (index needs to be repaired). |
| -10503 | Record # is out of range (during **GOTO RECORD**, or data file needs to be repaired). (see note 3) |
| -10502 | Invalid record structure (data file needs to be repaired). |
| -10501 | Invalid index page (index needs to be repaired). |
| -10500 | Invalid record address (database needs to be repaired). |
| -9999 | No more room to save the record. (see note 4) |
| -9998 | Duplicated key. |
| -9997 | Maximum number of records has been reached. |
| -9996 | Stack is full (too much recursion or nested calls). |
| -9995 | Demo limit has been reached. |
| -9994 | Serial communication interrupted by the user—user pressed Ctrl-Alt-Shift (Windows) or Command-Option-Shift (Mac OS). |
| -9993 | Menu bar is damaged (database needs to be repaired). |
| -9992 | Wrong password. |
| -9991 | Privileges error. |
| -9990 | Time-out error. |
| -9989 | Invalid structure (database needs to be repaired). |
| -9988 | The form cannot be loaded. Either the form or the structure is damaged. |
| -9987 | Some other records are already related to this record. |
| -9986 | Record locked during an automatic deletion action. |
| -9985 | Recursive integrity. |
| -9984 | Transaction has been cancelled because of a duplicated index key error. |
| -9983 | The same external package is installed twice. |
| -9982 | The record was not loaded because it is not in the selection on the workstation. |
| -9981 | Invalid field name/field number definition table sent by the workstation. |
| -9980 | The file cannot be created because the structure is locked. |

| | |
|---|---|
| -9979 | Unknown user. |
| -9978 | Bad user password. |
| -9977 | The selection does not exist. |
| -9976 | Backup in progress; no modification allowed. |
| -9975 | Transaction index page could not be loaded. |
| -9974 | Record has already been deleted. |
| -9973 | The TRIC resources are not the same. |
| -9972 | Table number is out of range requested by workstation. |
| -9971 | Field number is out of range requested by workstation. |
| -9970 | Field is not indexed. |
| -9969 | Invalid field type requested by workstation. |
| -9968 | Invalid selected record number requested by workstation. |
| -9967 | The record could not be modified because it could not be loaded. |
| -9966 | Invalid type requested by a workstation. |
| -9965 | Bad search definition table sent by a workstation. |
| -9964 | Bad sort definition table sent by a workstation. |
| -9963 | Invalid record number requested by a workstation. |
| -9962 | The backup cannot be run because the server is shutting down. |
| -9961 | The backup process is not currently running. |
| -9960 | 4D Backup is not installed on the server. |
| -9959 | The backup process has already been started by another user or process. |
| -9958 | The process could not be started. |
| -9957 | The choice list is locked. |
| -9956 | Versions of 4D Client and 4D Server are different. |
| -9955 | QuickTime is not installed. |
| -9954 | There is no current record. |
| -9953 | There is no Log file. |
| -9952 | Invalid data segment header. |
| -9951 | This field has no relation. |
| -9950 | Invalid data segment number. |
| -9949 | License or privilege error. |
| -9948 | A modal dialog is activated. |
| -9947 | The “Allow 4D Open connections” check box has not been selected. |
| -9946 | Unable to clear the named selection because it does not exist. |
| -9945 | CD-ROM 4D Runtime error; writing operations are not allowed. |
| -9944 | The user does not belong to the 4D Open access group. |
| -9943 | 4D Connectivity Plug-ins version error. |
| -9942 | 4D Client licensing scheme is incompatible with this version of 4D Server. |
| -9941 | Unknown EX_GESTALT selector. |
| -9940 | 4D Extension initialization failed. |
| -9939 | External routine not found. |
| -9938 | The current record has been changed from within the trigger. |
| -9937 | Password System is locked by another user. |
| -9936 | External password code does not match to the database one |
| -9935 | The XML file is not valid or is not well-formed. |
| -9934 | The XML file is not well-formed. |
| -9933 | The XML file is not valid. |
| -9932 | The XML DLL is not loaded. |
| -9931 | The index for this attribute is invalid. |
| -9930 | There is no attribute with this name for this element. |
| -9929 | The index for this element is invalid. |
| -9928 | The name of the element is unknown. |

-9927 The referenced element is not the “root”.
-9926 The referenced element is invalid.
-9925 The referenced element is null.
-9924 The file must be opened in read only
-9923 The attribute name is not valid
-9922 Missing the parameter value in the attributes definition
-9921 Attempt to write a XML Prolog on a non-empty document
-9920 The type of the node is invalid
-9919 This encoding is not supported.
-9918 The name of the element is invalid.
-9917 The type of the array passed in parameter is invalid.
-9916 The element is not open.
-9915 The document's reference is invalid.
-9914 Internal fault.
-9913 Network fault.
-9912 HTTP fault.
-9911 Parser fault.
-9910 Soap fault.
-9909 No window available to run the form.
-9855 Invalid parameter number 5.
-9854 Invalid parameter number 4.
-9853 Invalid parameter number 3.
-9852 Invalid parameter number 2.
-9851 Invalid parameter number 1.
-9850 Invalid area parameter passed to an external command.
-9803 Unnamed object found in the form "{form}".
-9802 Object path not unique: {path}. Verify the application with the MSC.
-9801 Unable to open the method: {path}
-9800 One of the processes modified the access rights.
-9799 Error during preview initialization.
-9778 An error occurred while loading the dictionary
-9777 The method language does not match the application localization: {path}
-9776 Unable to create method: {path}
-9775 Unable to write method code: {path}
-9774 Unable to read method code: {path}
-9773 Unable to write method comments: {path}
-9772 Unable to read method comments: {path}
-9771 Unable to write method properties: {path}
-9770 Unable to read method properties: {path}
-9769 Invalid method property: {path}
-9768 Invalid object path: {path}
-9767 Cannot update methods. One or more resources are locked.
-9766 The method is currently being edited.
-9765 One or more comments are currently being edited.
-9764 The comment is currently being edited.
-9763 The command cannot be executed in a component.
-9762 The command cannot be executed in a compiled database.
-9761 Invalid object type.
-9760 It is not possible to move or copy the XML node to the specified position.
-9759 The Object Library could not be opened.
-9758 The user form already exists.
-9757 The user form does not exist.

| | |
|---|---|
| -9756 | There is no user structure file. |
| -9755 | The user form does not have a name. |
| -9754 | This command cannot be used from a dialog window. |
| -9753 | The source form does not exist. |
| -9752 | The user form cannot be created. |
| -9751 | The source form is not accessible by the user. |
| -9750 | The source form is not editable. |
| -1 | Invalid table number requested by a Plug-In |
| 1001 | No column to import in table {TableName} |
| 1002 | Wrong table to import |
| 1003 | No column to export in table {TableName} |
| 1004 | Wrong table to export |
| 1006 | Program interrupted by user—user pressed Alt-click (Windows) or Option-click (Mac OS) |
| 1006 | Cannot save structure of database {BaseName} |
| 1007 | Cannot create data file of database: {BaseName} |
| 1008 | Wrong data segment in database {BaseName} |
| 1009 | Memory is full |
| 1010 | Cannot load a table definition of database {BaseName} |
| 1011 | Cannot open data file of database {BaseName} |
| 1012 | Data segment is full in database: {BaseName} |
| 1013 | Cannot save data segment in database {BaseName} |
| 1014 | Cannot read data segment of database {BaseName} |
| 1015 | Wrong database header in database {BaseName} |
| 1016 | Cannot create table in database {BaseName} |
| 1017 | Cannot read index list of database {BaseName} |
| 1018 | Cannot write index list of database {BaseName} |
| 1019 | Wrong table reference in database {BaseName} |
| 1020 | Wrong field reference in database {BaseName} |
| 1021 | Invalid index type in database {BaseName} |
| 1022 | Invalid field name in table {TableName} of database {BaseName} |
| 1023 | Invalid database name |
| 1024 | Cannot open structure of database {BaseName} |
| 1025 | Cannot create structure of database {BaseName} |
| 1026 | Cannot load bit selection of database {BaseName} |
| 1027 | Cannot load set of database {BaseName} |
| 1028 | Cannot modify set of database {BaseName} |
| 1029 | Cannot save set of database {BaseName} |
| 1030 | Cannot save BLOB {BlobNum} in table {TableName} of database {BaseName} |
| 1031 | Cannot load BLOB {BlobNum} in table {TableName} of database {BaseName} |
| 1032 | Cannot allocate BLOB {BlobNum} in table {TableName} of database {BaseName} |
| 1033 | Cannot load data bit table of database {BaseName} |
| 1034 | Cannot save data bit table of database {BaseName} |
| 1035 | Cannot load table of data bit tables of database {BaseName} |
| 1036 | Cannot save table of data bit tables in database {BaseName} |
| 1037 | Cannot close data segment of database {BaseName} |
| 1038 | Cannot delete data segment of database {BaseName} |
| 1039 | Cannot open data segment of database {BaseName} |
| 1040 | Cannot create data segment of database {BaseName} |
| 1041 | Cannot allocate space in data segment |
| 1042 | Cannot free space in data segment of database {BaseName} |
| 1043 | File is write protected |
| 1044 | Cannot access field in record {RecNum} in table {TableName} of database {BaseName} |

1045 Field definition code is missing in record {RecNum} in table {TableName} of database {BaseName}
1046 Cannot save record {RecNum} in table {TableName} of database {BaseName}
1047 Cannot load record {RecNum} in table {TableName} of database {BaseName}
1048 Cannot allocate record in table {TableName} of database {BaseName}
1049 Cannot update record {RecNum} in table {TableName} of database {BaseName}
1050 Wrong header for record {RecNum} in table {TableName} of database {BaseName}
1051 Cannot save definition of table {TableName} of database {BaseName}
1052 Cannot update definition of table {TableName} of database {BaseName}
1053 Field name already exists
1055 Cannot update index values while saving a record in table {TableName} of database {BaseName}
1056 Cannot update BLOBs while saving a record in table {TableName} of database {BaseName}
1057 Cannot delete BLOBs while saving or deleting a record in table {TableName} of database {BaseName}
1058 Cannot add field in table {TableName} of database {BaseName}
1059 Cannot allocate table in memory
1061 Cannot allocate record in memory
1062 Cannot completely delete table {TableName} of database {BaseName}
1063 Cannot lock record {RecNum} in table {TableName} of database {BaseName}
1064 Cannot unlock record {RecNum} in table {TableName} of database {BaseName}
1065 Cannot delete record {RecNum} in table {TableName} of database {BaseName}
1066 Record {RecNum} is locked in table {TableName} of database {BaseName}
1067 Sequential search could not be completed in table {TableName} of database {BaseName}<
1068 Cannot save header for table {TableName} of database {BaseName}<
1069 Cannot import data in table {TableName} of database {BaseName}
1070 Cannot load index header of database {BaseName}
1071 Cannot save header for index {IndexName} of database {BaseName}
1072 Cannot get page address for index {IndexName} of database {BaseName}
1073 Cannot set page address for index {IndexName} of database {BaseName}
1074 Cannot drop index {IndexName} of database {BaseName}
1075 Cannot sort index {IndexName} of database {BaseName}
1076 Cannot load page for index {IndexName} of database {BaseName}
1077 Cannot save page for index {IndexName} of database {BaseName}
1078 Cannot insert key into index {IndexName} of database {BaseName}
1079 Cannot delete key from index {IndexName} of database {BaseName}
1080 Cannot completely delete index {IndexName} of database {BaseName}
1081 Cannot complete scan on index {IndexName} of database {BaseName}
1082 Cannot complete sort of index {IndexName} of database {BaseName}
1083 Cannot insert key into page of index {IndexName} of database {BaseName}
1084 Cannot delete key from page of index {IndexName} of database {BaseName}
1085 Cannot load index cluster of database {BaseName}
1086 Cannot add to index cluster of database {BaseName}
1087 Cannot delete from index cluster of database {BaseName}
1088 Index {IndexName} is invalid
1089 SQL syntax error (Obsolete)
1090 SQL token not found (Obsolete)
1091 Not implemented
1092 Cannot register code
1093 Operation in progress cancelled by user
1094 Transaction conflict
1095 Invalid table name in database {BaseName}
1096 Table {TableName} is locked in database {BaseName}
1097 Database {BaseName} is locked
1098 Data address is invalid in database {BaseName}

| | |
|---|---|
| 1099 | Record is empty |
| 1100 | Wrong source field |
| 1101 | Wrong destination field |
| 1102 | Invalid relation name |
| 1103 | Field types do not match |
| 1104 | Relation is empty< |
| 1105 | Cannot load relations list from database {BaseName} |
| 1106 | Cannot save relations list into database {BaseName} |
| 1107 | Query and lock not completed; at least one record locked somewhere else |
| 1108 | Invalid record |
| 1109 | Wrong record ID |
| 1110 | Relation already exists in database {BaseName} |
| 1111 | Index already exists in database {BaseName} |
| 1112 | Wrong comparison operator |
| 1113 | End of data buffer |
| 1114 | Wrong DB4D version number |
| 1115 | Duplicated key |
| 1116 | Mandatory field is Null in record {RecNum} in table {TableName} |
| 1117 | Cannot set field to mandatory for table {TableName} |
| 1118 | Cannot get exclusive access to table {TableName} |
| 1119 | Cannot check referential integrity of table {TableName} |
| 1120 | Reference integrity: some foreign keys still exist matching primary key on record {RecNum} in table {TableName} |
| 1121 | Cannot delete all records in selection of table {TableName} |
| 1122 | BLOB is Null |
| 1123 | Invalid database context |
| 1124 | Invalid relation reference |
| 1125 | Invalid record name in table {TableName} |
| 1126 | Wrong field type |
| 1127 | Cannot load extra properties |
| 1128 | Cannot save extra properties |
| 1129 | Subrecord ID is out of range |
| 1130 | Duplicate name for index {IndexName} in database {BaseName} |
| 1131 | Invalid name for index {IndexName} in database {BaseName} |
| 1132 | Wrong key value on index {IndexName} |
| 1133 | Wrong type for index {IndexName} |
| 1134 | Invalid accessor |
| 1135 | Accessor is read only |
| 1136 | Null value not accepted |
| 1137 | THIS is Null |
| 1138 | Selection is Null |
| 1139 | Database {BaseName} is write protected |
| 1140 | Database {BaseName} is closing |
| 1141 | Invalid transaction |
| 1142 | Array limit is exceeded |
| 1143 | Creator of array values is missing |
| 1144 | Cannot build selection of table {TableName} |
| 1145 | Busy object |
| 1146 | Data file does not match structure file |
| 1147 | Cannot start listener |
| 1148 | Cannot start server |
| 1149 | No listener |
| 1150 | Task is dying |

| | |
|---|---|
| 1151 | Invalid request tag |
| 1152 | Invalid context ID |
| 1153 | Not enough temporary space on disk |
| 1154 | Data set is Null |
| 1155 | No primary key matching the foreign key |
| 1156 | Type of field {FieldName} of table {TableName} does not support being set to Unique |
| 1157 | Type of field {FieldName} of table {TableName} does not support being set to NEVER NULL |
| 1158 | Cannot alter primary key definition of table {TableName} |
| 1159 | Maximum number of records has been reached for table {TableName} |
| 1160 | Maximum number of BLOBs has been reached for table {TableName} |
| 1161 | Indice is out of range |
| 1162 | Query is invalid |
| 1163 | Record is NULL |
| 1164 | Object is NULL |
| 1165 | Wrong owner for this object |
| 1166 | Object was not locked |
| 1167 | Object is locked by another context |
| 1168 | Internal error on remote connection |
| 1169 | Invalid table number |
| 1170 | Invalid field number |
| 1171 | Invalid Database ID |
| 1172 | Invalid Parameter |
| 1173 | Cache Flushing did not complete |
| 1174 | Data Flushing did not complete |
| 1175 | Structure Flushing did not complete |
| 1176 | Log file is invalid for database {BaseName} |
| 1177 | Log file cannot be found for database {BaseName} |
| 1178 | Last operation in log file doest not match the one in database {BaseName} |
| 1179 | Log file does not match database {BaseName} |
| 1180 | Data Table has been deleted |
| 1181 | Keys are not unique in index {IndexName} |
| 1182 | Cannot create log file for database {DataBase} |
| 1183 | Cannot write into log file of database {DataBase} |
| 1184 | Cannot drop table {TableName} in database {DataBase} |
| 1185 | Remote database cannot be opened< |
| 1186 | Log file cannot be integrated into database {DataBase} |
| 1187 | Internal Computation on Set cannot be completed |
| 1188 | Array cannot be saved |
| 1189 | Array cannot be loaded |
| 1190 | Sequence number header cannot be loaded |
| 1191 | Cannot select record |
| 1192 | Record cannot be created |
| 1193 | Cannot complete selection to array in table {TableName} of database {BaseName} |
| 1194 | Cannot complete array to selection in table {TableName} of database {BaseName} |
| 1195 | Cannot complete sequential sort |
| 1196 | Selection cannot be locked |
| 1197 | Index key cannot be loaded |
| 1198 | Index key cannot be saved |
| 1199 | Index key cannot be built |
| 1200 | Query cannot be completed |
| 1201 | Query cannot be analyzed |
| 1202 | Formula could not be processed on this column |

| | |
|---|---|
| 1203 | Query could not be completed |
| 1204 | Query could not be analyzed |
| 1205 | Could not get all distinct values of table {TableName} of database {BaseName} |
| 1206 | Cannot build array of values in table {TableName} of database {BaseName} |
| 1207 | Selection cannot be loaded |
| 1208 | Cannot send data |
| 1209 | Cannot receive request reply |
| 1210 | Cannot send request |
| 1211 | Cannot create connection |
| 1212 | Index {IndexName} cannot be quickly created in database {BaseName} |
| 1213 | Cannot build distinct index keys |
| 1214 | Selection cannot be sorted in table {TableName} of database {BaseName} |
| 1215 | Address table cannot loaded |
| 1216 | Address table cannot modified |
| 1217 | Cannot allocate new entry into address table |
| 1218 | Cannot allocate free entry from address table |
| 1219 | Transaction temporary record cannot be saved |
| 1220 | Transaction temporary blob cannot be saved |
| 1221 | Transaction temporary record cannot be loaded |
| 1222 | Transaction temporary blob cannot be loaded |
| 1223 | Transaction cannot be started |
| 1224 | Transaction cannot be committed |
| 1225 | Cannot get extra property |
| 1226 | Cannot set extra property |
| 1227 | Table name is already used |
| 1228 | Cannot get list of NULL keys from index {IndexName} |
| 1229 | Cannot modify list of NULL keys of index {IndexName} |
| 1230 | Invalid key for index {IndexName} |
| 1231 | Cannot set log file |
| 1232 | Context is NULL |
| 1233 | Database {BaseName} cannot be locked |
| 1234 | Wrong field reference in record# {RecNum} of table: {TableName} of database: {BaseName} |
| 1235 | Wrong field reference in table: {TableName} of database: {BaseName} |
| 1236 | Cannot read data from temporary transaction file |
| 1237 | Cartesian Product failed |
| 1238 | Cannot merge selections |
| 1239 | The format of the {BaseName} database cannot be upgraded in read only mode |
| 1240 | Wrong Header |
| 1241 | Wrong CheckSum |
| 1243 | Cannot load data tables of database: {BaseName} |
| 1244 | The list of foreign key constraints is not empty for table: {TableName} of database: {BaseName} |
| 1245 | Address entry is not empty |
| 1246 | Cannot pre-allocate address |
| 1247 | Cannot update new record in table {TableName} of database {BaseName} |
| 1248 | Cannot save new record in table {TableName} of database {BaseName} |
| 1249 | Cannot save subrecord |
| 1250 | Cannot save record |
| 1251 | Cannot lock structure object definition |
| 1252 | Cannot unlock structure object definition |
| 1253 | Invalid relation number |
| 1254 | Circular Reference in records address table of table {TableName} of database {BaseName} |
| 1255 | Circular Reference in blobs address table of table {TableName} of database {BaseName} |

| | |
|---|---|
| 1256 | Duplicated Schema Name in database {BaseName} |
| 1257 | Schema cannot be saved in database {BaseName} |
| 1258 | Schema cannot be deleted in database {BaseName} |
| 1259 | Schema cannot be renamed in database {BaseName} |
| 1260 | Selected log file is too recent for database {BaseName} |
| 1261 | Selected log file is too old for database {BaseName} |
| 1262 | Some DataTables do not have matching table definitions for database {BaseName} |
| 1263 | Given stamp does not match current one for record# {RecNum} of table {TableName} |
| 1264 | A primary key is needed and missing in table {TableName} |
| 1265 | Invalid field |
| 1266 | Unknown field type (this version of 4D is perhaps too old) |
| 1267 | Stack Overflow |
| 1268 | DataTable cannot be rebuilt for database {BaseName} |
| 1269 | DataTable cannot be found |
| 1270 | Missing structure could not be rebuilt |
| 1271 | Invalid REST request handler |
| 1272 | Some records are still locked in table {tableName} in database {BaseName} |
| 1273 | Cannot drop table {TableName} in database {BaseName} |
| 1274 | The current journal file of database {BaseName} is not available. For data security reasons, write operations are stopped. Please contact your administrator as soon as possible. |
| 1275 | "(" is missing at the beginning of the Javascript statement in this query. |
| 1276 | Datastream has an invalid header. |
| 1277 | The database journal integration failed at entry# {p1}. |
| 1278 | Javascript code not allowed in the query. |
| 1300 | Invalid Selection Type |
| 1301 | Array is too big |
| 1302 | Array size does not match |
| 1303 | Invalid Selection ID |
| 1304 | Invalid Selection Part |
| 4001 | Invalid table number requested by a Plug-In |
| 4002 | Invalid record number requested by a Plug-In |
| 4003 | Invalid field number requested by a Plug-In |
| 4004 | Access to a table's current record requested by a Plug-in while there is no current record |

**Notes**

1. While some of the errors listed reflect serious problems, i.e., *-10502 Invalid record structure (data file needs to be repaired)*, other errors may occur on a regular basis and can be managed using an **ON ERR CALL** project method. For example, it is common to handle the error *–9998 Duplicated key* if your application offers opportunities to create duplicated values for a table that includes an indexed field whose Unique property is set.
2. Some of the errors listed never occur at the 4D language level. They can occur and be handled at a low level by database engine routines or when using 4D Backup or 4D Open.
3. The error *-10503 Record # is out of range* generally means that your code (for example, the **GOTO RECORD** command) is attempting to access a record that no longer exists (or has never existed). In certain more unusual cases, this error may mean that the database needs repairing.
4. The error *-9999 No more room to save the record* occurs when the data file of your database is full or located on a full volume. This error can also be generated if the data file is locked or located on a locked volume.

# SQL Engine Errors (1001 -> 3018)

The SQL engine of 4D returns specific errors which are listed below. These errors can be intercepted using an error-handling method installed by the **ON ERR CALL** command and analyzed using the **GET LAST ERROR STACK** command.

## Generic errors

| | |
|---|---|
| 1001 | INVALID ARGUMENT |
| 1002 | INVALID INTERNAL STATE |
| 1003 | SQL SERVER IS NOT RUNNING |
| 1004 | Access denied |
| 1005 | FAILED TO LOCK SYNCHRONIZATION PRIMITIVE |
| 1006 | FAILED TO UNLOCK SYNCHRONIZATION PRIMITIVE |
| 1007 | SQL SERVER IS NOT AVAILABLE |
| 1008 | COMPONENT BRIDGE IS NOT AVAILABLE |
| 1009 | REMOTE SQL SERVER IS NOT AVAILABLE |
| 1010 | EXECUTION INTERRUPTED BY USER |

## Semantic errors

| | |
|---|---|
| 1101 | Table '{key1}' does not exist in the database. |
| 1102 | Column '{key1}' does not exist. |
| 1103 | Table '{key1}' is not declared in the FROM clause. |
| 1104 | Column name reference '{key1}' is ambiguous. |
| 1105 | Table alias '{key1}' is the same as table name. |
| 1106 | Duplicate table alias - '{key1}'. |
| 1107 | Duplicate table in the FROM clause - '{key1}'. |
| 1108 | Operation {key1} {key2} {key3} is not type safe. |
| 1109 | Invalid ORDER BY index - {key1}. |
| 1110 | Function {key1} expects one parameter, not {key2}. |
| 1111 | Parameter {key1} of type {key2} in function call {key3} is not implicitly convertible to {key4}. |
| 1112 | Unknown function - {key1}. |
| 1113 | Division by zero. |
| 1114 | Sorting by indexed item in the SELECT list is not allowed - ORDER BY item {key1}. |
| 1115 | DISTINCT NOT ALLOWED |
| 1116 | Nested aggregate functions are not allowed in the aggregate function {key1}. |
| 1117 | Column function is not allowed. |
| 1118 | Cannot mix column and scalar operations. |
| 1119 | Invalid GROUP BY index - {key1}. |
| 1120 | GROUP BY index is not allowed. |
| 1121 | GROUP BY is not allowed with 'SELECT * FROM ...'. |
| 1122 | HAVING is not an aggregate expression. |
| 1123 | Column '{key1}' is not a grouping column and cannot be used in the ORDER BY clause. |
| 1124 | Cannot mix {key1} and {key2} types in the IN predicate. |
| 1125 | Escape sequence '{key1}' in the LIKE predicate is too long. It must be exactly one character. |
| 1126 | Bad escape character - '{key1}'. |
| 1127 | Unknown escape sequence - '{key1}'. |
| 1128 | Column references from more than one query in aggregate function {key1} are not allowed. |
| 1129 | Scalar item in the SELECT list is not allowed when GROUP BY clause is present. |
| 1130 | Sub-query produces more than one column. |
| 1131 | Subquery must return one row at the most but instead it returns {key1}. |
| 1132 | INSERT value count {key1} does not match column count {key2}. |
| 1133 | Duplicate column reference in the INSERT list - '{key1}'. |
| 1134 | Column '{key1}' does not allow NULL values. |
| 1135 | Duplicate column reference in the UPDATE list - '{key1}'. |
| 1136 | Table '{key1}' already exists. |
| 1137 | Duplicate column '{key1}' in the CREATE TABLE command. |
| 1138 | DUPLICATE COLUMN IN COLUMN LIST |
| 1139 | More than one primary key is not allowed. |
| 1140 | Ambiguous foreign key name - '{key1}'. |
| 1141 | Column count {key1} in the child table does not match column count {key2} in the parent table of the foreign key definition. |
| 1142 | Column type mismatch in the foreign key definition. Cannot relate {key1} in child table to {key2} in parent table. |
| 1143 | Failed to find matching column in parent table for '{key1}' column in child table. |
| 1144 | UPDATE and DELETE constraints must be the same. |
| 1145 | FOREIGN KEY DOES NOT EXIST |
| 1146 | Invalid LIMIT value in SELECT command - {key1}. |
| 1147 | Invalid OFFSET value in SELECT command - {key1}. |
| 1148 | Primary key does not exist in table '{key1}'. |
| 1149 | FAILED TO CREATE FOREIGN KEY |
| 1150 | Column '{key1}' is not part of a primary key. |
| 1151 | FIELD IS NOT UPDATEABLE |
| 1152 | FOUND VIEW COLUMN |

| | |
|---|---|
| 1153 | Bad data type length '{key1}'. |
| 1154 | EXPECTED EXECUTE IMMEDIATE COMMAND |
| 1155 | INDEX ALREADY EXISTS |
| 1156 | Auto-increment option is not allowed for column '{key1}' of type {key2}. |
| 1157 | SCHEMA ALREADY EXISTS |
| 1158 | SCHEMA DOES NOT EXIST |
| 1159 | Cannot drop system schema |
| 1160 | CHARACTER ENCODING NOT ALLOWED |

### Implementation errors

| | |
|---|---|
| 1203 | FUNCTIONALITY IS NOT IMPLEMENTED |
| 1204 | Failed to create record {key1}. |
| 1205 | Failed to update field '{key1}'. |
| 1206 | Failed to delete record '{key1}'. |
| 1207 | NO MORE JOIN SEEDS POSSIBLE |
| 1208 | FAILED TO CREATE TABLE |
| 1209 | FAILED TO DROP TABLE |
| 1210 | CANT BUILD BTREE FOR ZERO RECORDS |
| 1211 | COMMAND COUNT GREATER THAN ALLOWED |
| 1212 | FAILED TO CREATE DATABASE |
| 1213 | FAILED TO DROP COLUMN |
| 1214 | VALUE IS OUT OF BOUNDS |
| 1215 | FAILED TO STOP SQL_SERVER |
| 1216 | FAILED TO LOCALIZE |
| 1217 | Failed to lock table for reading. |
| 1218 | FAILED TO LOCK TABLE FOR WRITING |
| 1219 | TABLE STRUCTURE STAMP CHANGED |
| 1220 | FAILED TO LOAD RECORD |
| 1221 | FAILED TO LOCK RECORD FOR WRITING |
| 1222 | FAILED TO PUT SQL LOCK ON A TABLE |
| 1223 | FAILED TO RETAIN COOPERATIVE TASK |
| 1224 | FAILED TO LOAD INFILE |

### Parsing error

| | |
|---|---|
| 1301 | PARSING FAILED |

### Runtime language access errors

| | |
|---|---|
| 1401 | COMMAND NOT SPECIFIED |
| 1402 | ALREADY LOGGED IN |
| 1403 | SESSION DOES NOT EXIST |
| 1404 | UNKNOWN BIND ENTITY |
| 1405 | INCOMPATIBLE BIND ENTITIES |
| 1406 | REQUEST RESULT NOT AVAILABLE |
| 1407 | BINDING LOAD FAILED |
| 1408 | COULD NOT RECOVER FROM PREVIOUS ERRORS |
| 1409 | NO OPEN STATEMENT |
| 1410 | RESULT EOF |
| 1411 | BOUND VALUE IS NULL |
| 1412 | STATEMENT ALREADY OPENED |
| 1413 | FAILED TO GET PARAMETER VALUE |
| 1414 | INCOMPATIBLE PARAMETER ENTITIES |
| 1415 | Query parameter is either not allowed or was not provided. |
| 1416 | COLUMN REFERENCE PARAMETERS FROM DIFFERENT TABLES |
| 1417 | EMPTY STATEMENT |
| 1418 | FAILED TO UPDATE VARIABLE |
| 1419 | FAILED TO GET TABLE REFERENCE |
| 1420 | FAILED TO GET TABLE CONTEXT |
| 1421 | COLUMNS NOT ALLOWED |
| 1422 | INVALID COMMAND COUNT |
| 1423 | INTO CLAUSE NOT ALLOWED |
| 1424 | EXECUTE IMMEDIATE NOT ALLOWED |
| 1425 | ARRAY NOT ALLOWED IN EXECUTE IMMEDIATE |
| 1426 | COLUMN NOT ALLOWED IN EXECUTE IMMEDIATE |
| 1427 | NESTED BEGIN END SQL NOT ALLOWED |
| 1428 | RESULT IS NOT A SELECTION |
| 1429 | INTO ITEM IS NOT A VARIABLE |
| 1430 | VARIABLE WAS NOT FOUND |
| 1431 | PTR OF PTR NOT ALLOWED |
| 1432 | POINTER OF UNKNOWN TYPE |

### Date formatting errors

| | |
|---|---|
| 1501 | SEPARATOR_EXPECTED |
| 1502 | FAILED TO PARSE DAY OF MONTH |
| 1503 | FAILED TO PARSE MONTH |
| 1504 | FAILED TO PARSE YEAR |
| 1505 | FAILED TO PARSE HOUR |
| 1506 | FAILED TO PARSE MINUTE |
| 1507 | FAILED TO PARSE SECOND |
| 1508 | FAILED TO PARSE MILLISECOND |
| 1509 | INVALID AM PM USAGE |
| 1510 | FAILED TO PARSE TIME ZONE |
| 1511 | UNEXPECTED CHARACTER |
| 1512 | Failed to parse timestamp. |
| 1513 | Failed to parse duration. |
| 1551 | FAILED TO PARSE DATE FORMAT |

### Lexer errors

| | |
|---|---|
| 1601 | NULL INPUT STRING |
| 1602 | NON TERMINATED STRING |
| 1603 | NON TERMINATED COMMENT |
| 1604 | INVALID NUMBER |
| 1605 | UNKNOWN START OF TOKEN |
| 1606 | NON TERMINATED NAME/* closing ']' is missing |
| 1607 | NO VALID TOKENS |

### Validation errors for error stack - state errors that follow direct errors

| | |
|---|---|
| 1701 | Failed to validate table '{key1}'. |
| 1702 | Failed to validate FROM clause. |
| 1703 | Failed to validate GROUP BY clause. |
| 1704 | Failed to validate SELECT list. |
| 1705 | Failed to validate WHERE clause. |
| 1706 | Failed to validate ORDER BY clause. |
| 1707 | Failed to validate HAVING clause. |
| 1708 | Failed to validate COMPARISON predicate. |
| 1709 | Failed to validate BETWEEN predicate. |
| 1710 | Failed to validate IN predicate. |
| 1711 | Failed to validate LIKE predicate. |
| 1712 | Failed to validate ALL ANY predicate. |
| 1713 | Failed to validate EXISTS predicate. |
| 1714 | Failed to validate IS NULL predicate. |
| 1715 | Failed to validate subquery. |
| 1716 | Failed to validate SELECT item {key1}. |
| 1717 | Failed to validate column '{key1}'. |
| 1718 | Failed to validate function '{key1}'. |
| 1719 | Failed to validate CASE expression. |
| 1720 | Failed to validate command parameter. |
| 1721 | Failed to validate function parameter {key1}. |
| 1722 | Failed to validate INSERT item {key1}. |
| 1723 | Failed to validate UPDATE item {key1}. |
| 1724 | Failed to validate column list. |
| 1725 | Failed to validate foreign key. |
| 1726 | Failed to validate SELECT command. |
| 1727 | Failed to validate INSERT command. |
| 1728 | Failed to validate DELETE command. |
| 1729 | Failed to validate UPDATE command. |
| 1730 | Failed to validate CREATE TABLE command. |
| 1731 | Failed to validate DROP TABLE command. |
| 1732 | Failed to validate ALTER TABLE command. |
| 1733 | Failed to validate CREATE INDEX command. |
| 1734 | Failed to validate LOCK TABLE command. |
| 1735 | Failed to calculate LIKE predicate pattern. |

### Execution errors for error stack - state errors that follow direct errors

| | |
|---|---|
| 1801 | Failed to execute SELECT command. |
| 1802 | Failed to execute INSERT command. |
| 1803 | Failed to execute DELETE command. |
| 1804 | Failed to execute UPDATE command. |
| 1805 | Failed to execute CREATE TABLE command. |
| 1806 | Failed to execute DROP TABLE command. |
| 1807 | Failed to execute CREATE DATABASE command. |
| 1808 | Failed to execute ALTER TABLE command. |
| 1809 | Failed to execute CREATE INDEX command. |
| 1810 | Failed to execute DROP INDEX command. |
| 1811 | Failed to execute LOCK TABLE command. |
| 1812 | Failed to execute TRANSACTION command. |
| 1813 | Failed to execute WHERE clause. |
| 1814 | Failed to execute GROUP BY clause. |
| 1815 | Failed to execute HAVING clause. |
| 1816 | Failed to aggregate. |
| 1817 | Failed to execute DISTINCT. |
| 1818 | Failed to execute ORDER BY clause. |
| 1819 | Failed to build DB4D query. |
| 1820 | Failed to calculate comparison predicate. |
| 1821 | Failed to execute subquery. |
| 1822 | Failed to calculate BETWEEN predicate. |
| 1823 | Failed to calculate IN predicate. |
| 1824 | Failed to calculate ALL/ANY predicate. |
| 1825 | Failed to calculate LIKE predicate. |
| 1826 | Failed to calculate EXISTS predicate. |
| 1827 | Failed to calculate NULL predicate. |
| 1828 | Failed to perform arithmetic operation. |
| 1829 | Failed to calculate function call '{key1}'. |
| 1830 | Failed to calculate case expression. |
| 1831 | Failed to calculate function parameter '{key1}'. |
| 1832 | Failed to calculate 4D function call. |
| 1833 | Failed to sort while executing ORDER BY clause. |
| 1834 | Failed to calculate record hash. |
| 1835 | Failed to compare records. |
| 1836 | Failed to calculate INSERT value {key1}. |
| 1837 | DB4D QUERY FAILED |
| 1838 | FAILED TO EXECUTE ALTER SCHEMA COMMAND |
| 1839 | FAILED TO EXECUTE GRANT COMMAND |

### Cacheable errors

| | |
|---|---|
| 2000 | CACHEABLE NOT INITIALIZED |
| 2001 | VALUE ALREADY CACHED |
| 2002 | CACHED VALUE NOT FOUND |
| 2003 | CACHE IS FULL |
| 2004 | CACHING IS NOT POSSIBLE |

### Protocol errors

| | |
|---|---|
| 3000 | HEADER NOT FOUND |
| 3001 | UNKNOWN COMMAND |
| 3002 | ALREADY LOGGED IN |
| 3003 | NOT LOGGED IN |
| 3004 | UNKNOWN OUTPUT MODE |
| 3005 | INVALID STATEMENT ID |
| 3006 | UNKNOWN DATA TYPE |
| 3007 | STILL LOGGED IN |
| 3008 | SOCKET READ ERROR |
| 3009 | SOCKET WRITE ERROR |
| 3010 | BASE64 DECODING ERROR |
| 3011 | SESSION TIMEOUT |
| 3012 | FETCH TIMESTAMP ALREADY EXISTS |
| 3013 | BASE64 ENCODING ERROR |
| 3014 | INVALID HEADER TERMINATOR |
| 3015 | INVALID SESSION TICKET |
| 3016 | HEADER TOO LONG |
| 3017 | INVALID AGENT SIGNATURE |
| 3018 | UNEXPECTED HEADER VALUE |

## Network Errors (-10051 -> -10001)

---

The following table describes the errors that can occur with a network connection.

| Code | Description |
|---|---|
| -10051 | Incorrect value for network component option. |
| -10050 | Unknown network component option. |
| -10033 | Incorrect data size during read cycle. |
| -10031 | Desynchronization has occurred during the read cycle. |
| -10030 | Desynchronization has occurred during the write cycle. |
| -10021 | No server was found. |
| -10020 | No server was selected. |
| -10003 | Bad connection parameters. |
| -10002 | The connection for this process has been disrupted or the connection couldn't be established. |
| -10001 | The current connection to the database has been disrupted. |

## Backup Manager Errors (1401 -> 1421)

---

The following table lists the specific error codes generated by the backup and restore module of 4D.
You can retrieve these errors using a method installed via the **ON ERR CALL** command.

| Code | Description |
|---|---|
| 1401 | The maximum number of backup attempts has been reached; automatic backup is temporarily disabled. |
| 1403 | No log file. |
| 1404 | A transaction is opened in this process. |
| 1405 | The maximum timeout for transactions to end in a concurrent process has been reached. |
| 1406 | Backup canceled by user. |
| 1407 | Destination folder is not valid. |
| 1408 | Error during log file backup. |
| 1409 | Error during backup. |
| 1410 | Cannot find the backup file to be checked. |
| 1411 | Error during backup file check. |
| 1412 | Cannot find the log backup file to be checked. |
| 1413 | Error during log backup file check. |
| 1414 | This command can only be executed on 4D Server. |
| 1415 | Cannot back up log file; a critical operation is in progress. |
| 1416 | This log file does not correspond to the database opened. |
| 1417 | A log integration operation is already running. The backup cannot be launched. |
| 1420 | Integration aborted due to detection of locked records. |
| 1421 | This command cannot be used in a client/server environment. |

- Errors 1408 and 1409 generally come from a read error for files to be backed up or a write error during file backup.
- Errors 1411 and 1413 occur during checking of archives.
  When these errors occur, it may be prudent to first check the space remaining on the disk and the read-write access privileges.

## Client Update Errors (-10650 -> -10655)

---

The following table describes errors that may occur in the context of a client update:

| Code | Description |
|---|---|
| -10650 | Cannot download client update |
| -10651 | Cannot expand client update |
| -10652 | Update information not available ({path}) |
| -10653 | Cannot download update '{path}' |
| -10654 | Update information invalid |
| -10655 | Client update is not available |

## Updater Errors (-10603 -> -10613)

---

The following table describes updater errors that may occur:

| Code | Description |
|---|---|
| -10603 | Updater installation failed |
| -10604 | Cannot identify updater package {path} |
| -10605 | Error while copying updater package files |
| -10606 | Cannot disable current updater version |
| -10607 | Cannot create new updater install folder |
| -10608 | Cannot find folder containing software update {path} |
| -10609 | Cannot start software updater |
| -10611 | Cannot start software updater (invalid install) |
| -10612 | The running application cannot be updated with itself |
| -10613 | Cannot create two form windows of type toolbar |

## OS File Manager Errors (-124 -> -33)

---

The following table lists codes returned by the Operating System File Manager. These codes can be returned when you are using, for example, the System Documents commands (see the **System Documents** chapter). In this list, the word “file” indicates a document on disk and not a file in your database structure.

| Code | Description |
|---|---|
| -124 | Tried to access a disconnected shared volume. |
| -121 | An access path could not be created. |
| -120 | Tried to access a file by using a pathname that specifies a non existing directory. |
| -84 | There is a hardware problem with the disk (bad installation, incorrect formatting,...). |
| -64 | There is a hardware problem with the disk (bad installation, incorrect formatting,...). |
| -61 | Read/write permission doesn’t allow writing. |
| -60 | Bad master directory block. Your disk is damaged. |
| -58 | Error in the external file system. |
| -57 | Tried to work with a non-Macintosh disk. |
| -54 | Attempt to open locked file for writing. |
| -53 | Volume not on line. |
| -52 | Internal file manager error (position of file marker is lost). |
| -51 | Tried to access a document with an invalid document reference number. |
| -49 | Tried to open a file already open with write permission. |
| -48 | Tried to rename a file with the name of an already deleted file. |
| -47 | The file is already open, or the folder is not empty. |
| -46 | Volume is locked by an application. |
| -45 | The file is locked, or the pathname is not correct. |
| -44 | Volume is locked by a hardware setting. |
| -43 | File not found. |
| -42 | Too many files open at the same time. |
| -41 | Not enough memory to open a new file on the disk. |
| -40 | Attempt to position before start of file. |
| -39 | Logical end-of-file reached during read operation. |
| -38 | Tried to read or write to a file that is not open. |
| -37 | Bad filename or volume name. |
| -36 | I/O error. There is probably a bad block on the disk. |
| -35 | Specified volume doesn’t exist. |
| -34 | Disk is full. There is no more room available on the disk. |
| -33 | File directory full. You cannot create new files on disk. |

## OS Memory Manager Errors (-117 -> -108)

---

The following table lists the error codes returned by the Operating System Memory Manager.

| Code | Description |
|---|---|
| -117 | Internal Memory problem. Memory is probably logically corrupted.<br>Exit as soon as possible. Restart your machine and reopen the database. |
| -111 | Internal Memory problem. Memory is probably logically corrupted.<br>Exit as soon as possible. Restart your machine and reopen the database. (*) |
| -109 | Internal Memory problem. Memory is probably logically corrupted.<br>Exit as soon as possible. Restart your machine and reopen the database. |
| -108 | Not enough memory to perform an operation.<br>Give more memory to your 4D application. |

**Tip**: When allocating and working with large arrays, BLOBs, pictures, as well as sets (objects that can hold large amount of data), use an **ON ERR CALL**. project method to test the error -108.

(*) Error -111 can also occur when you attempt to read a value from a BLOB with an offset out of range. In this case, the error is minor and you do not need to terminate the working session. Just fix the offset you pass to the BLOB command.

## OS Printing Manager Errors (-8192 -> -1)

---

The following table lists the error codes returned by the Operating System Printing Manager. These codes can be returned during printing.

| Code | Description |
|---|---|
| -8192 | LaserWriter time-out |
| -8151 | The printer has been initialized with a different driver version |
| -8150 | A LaserWriter is not selected |
| -4101 | Printer busy or not connected |
| -4100 | Printer connection has been interrupted |
| -193 | Resource file not found |
| -128 | Printing interrupted by the user |
| -27 | Problem opening or closing connection with printer |
| -1 | Problem saving file to be printed |

## OS Resource Manager Errors (-196 -> -1)

---

The following table lists the error codes returned by the Operating System Resource Manager.

| Code | Description |
|---|---|
| -196 | Resource could not be deleted |
| -194 | Resource could not be added |
| -193 | Resource map is damaged (file needs to be repaired) |
| -192 | Resource not found |
| -1 | Resource file could not be opened |

## SANE NaN Errors (1 -> 255)

---

The following table lists the NaN codes returned by the Operating System. NaN is a Standard Apple Numeric Environment (SANE) representation which means “Not a Number.” NaN appears when an operation produces a result that is beyond SANE’s scope.

| Code | Description |
|---|---|
| 1 | Invalid square root |
| 2 | Invalid addition |
| 4 | Invalid division |
| 8 | Invalid multiplication |
| 9 | Invalid remainder |
| 17 | Converting an invalid ASCII string |
| 20 | Converting a Comp type number to floating-point |
| 21 | Creating a NaN with a zero code |
| 33 | Invalid argument to a trig function |
| 34 | Invalid argument to an inverse trig function |
| 36 | Invalid argument to a log function |
| 37 | Invalid argument to an xi or xy function |
| 38 | Invalid argument to a financial function |
| 255 | Uninitialized storage |

## OS Sound Manager Errors (-209 -> -203)

---

The following table lists the codes returned by the Operating System Sound Manager.

| Code | Description |
|---|---|
| -209 | The sound channel is busy |
| -207 | Not enough memory to perform the sound |
| -206 | The format of the sound resource is wrong |
| -205 | The sound channel is logically corrupted |
| -204 | The sound resource cannot be loaded |
| -203 | Too many sound commands |

## OS Serial Ports Manager Errors (-28)

---

The following table lists error codes returned by the Operating System Serial Ports Manager.

| Code | Description |
|---|---|
| -28 | There is no open serial port |

## Mac OS System Errors (4 -> 28)

---

The following table lists some of the Mac OS system errors. It is usually not possible to recover from these errors.

| Code | Description |
|---|---|
| 4 | Zero divide |
| 15 | Segment Loader Error: 4D failed in loading one of its own code segments.<br>You must allocate more memory to 4D. |
| 17 to 24 | A system package is missing. Check if your system directory has been correctly installed |
| 25 | Out of memory<br>You must allocate more memory to 4D. |
| 28 | Stack has moved into the application heap.<br>You must allocate more memory to 4D. |

## Miscellaneous Errors (-10700)

---

The following table describes miscellaneous errors that may occur:

| Code | Description |
| --- | --- |
| -10700 | Port must be in the 0 to 65535 range. |

# Character Codes

- Unicode Codes
- Function Key Codes

## Unicode Codes

---

In databases created with version 11 of 4D, the language as well as the database engine store and work natively with Unicode characters.

This facilitates the internationalization of 4D applications. Unicode is a standard unified character set that can handle practically every common language of the world. A character set is a character/number value correspondence table, for example “a”->1, “b”->2, “5”->15, “oe”->662, and so on. Whereas with ASCII, the basic number value is typically included between 1 and 127, with Unicode the upper limit exceeds 65,000, which means that nearly every character for all languages can be represented.

There are several ways to code the Unicode number values: UTF-16 codes them on 16-bit integers, UTF-32 uses 32-bit integers and UTF-8 uses 8-bit integers. 4D mainly uses UTF-16 (like Windows and Mac OS).

Sometimes, essentially for specific needs related to the Internet, 4D uses UTF-8 which has the advantage of being more compact and having better readability for common characters (a-z,0-9).

For more information about the Unicode standard, please refer, for example, to the following page:
http://en.wikipedia.org/wiki/Unicode

A list of Unicode codes:
http://en.wikipedia.org/wiki/List_of_Unicode_characters

**Warning:** In Unicode in 4D v11, the following character codes are reserved and must never be included in a text:
0
65534 (FFFE)
65535 (FFFF)

**Compatibility Note:** Databases created with a version of 4D prior to version 11 can function in ASCII compatibility mode. For more information, please refer to the **EXPORT TEXT** section.

## Function Key Codes

4D returns values for Function keys in the *KeyCode* system variable, which is used within project methods installed by the **ON EVENT CALL** command. These project methods are used to catch events.

The values for Function keys are not based on ASCII Codes. They are:

| Function key | KeyCode |
|---|---|
| F1 | -122 |
| F2 | -120 |
| F3 | -99 |
| F4 | -118 |
| F5 | -96 |
| F6 | -97 |
| F7 | -98 |
| F8 | -100 |
| F9 | -101 |
| F10 | -109 |
| F11 | -103 |
| F12 | -111 |
| F13 | -105 |
| F14 | -107 |
| F15 | -113 |

**Reminder**

The *KeyCode* system variable is to be used in a project method installed using **ON EVENT CALL**.

In addition to the function keys, the following table lists the values returned in *KeyCode* when you press one of the common keys, such as **Return** or **Enter**.

| Key | Code |
|---|---|
| Enter | 3 |
| Return | 13 |
| Backspace | 8 |
| Tab | 9 |
| Escape | 27 |
| Del | 127 |
| Help | 5 |
| Home | 1 |
| End | 4 |
| Page Up | 11 |
| Page Down | 12 |
| Left Arrow | 28 |
| Right Arrow | 29 |
| Up Arrow | 30 |
| Down Arrow | 31 |

# 4D Language Reference - Obsolete commands

16.0

*_o_ADD DATA SEGMENT*
*_o_ADD SUBRECORD*

- _o_ALL SUBRECORDS
- _o_APPLY TO SUBSELECTION
- _o_ARRAY STRING
- _o_ARRAY TO STRING LIST
- _o_Before subselection
- _o_C_GRAPH
- _o_C_INTEGER
- _o_C_STRING
- _o_Convert case
- _o_Create resource file
- _o_CREATE SUBRECORD
- _o_DATA SEGMENT LIST
- _o_DELETE RESOURCE
- _o_DELETE SUBRECORD
- _o_DISABLE BUTTON
- _o_During
- _o_ENABLE BUTTON
- _o_End subselection
- _o_FIRST SUBRECORD
- _o_Font name
- _o_Font number
- _o_Get component resource ID
- _o_Get platform interface
- _o_GRAPH TABLE
- _o_INTEGRATE LOG FILE
- _o_INVERT BACKGROUND
- _o_ISO to Mac
- _o_LAST SUBRECORD
- _o_Mac to ISO
- _o_Mac to Win
- _o_MODIFY SUBRECORD
- _o_NEXT SUBRECORD
- _o_Open external window
- _o_ORDER SUBRECORDS BY
- _o_PICTURE TYPE LIST
- _o_PREVIOUS SUBRECORD
- _o_QT COMPRESS PICTURE
- _o_QT COMPRESS PICTURE FILE
- _o_QT LOAD COMPRESS PICTURE FROM FILE
- _o_QUERY SUBRECORDS
- _o_Records in subselection
- _o_REDRAW LIST
- _o_SAVE PICTURE TO FILE
- _o_SET CGI EXECUTABLE
- _o_SET PICTURE RESOURCE
- _o_SET PLATFORM INTERFACE
- _o_SET RESOURCE
- _o_SET RESOURCE NAME
- _o_SET RESOURCE PROPERTIES
- _o_SET STRING RESOURCE
- _o_SET TEXT RESOURCE
- _o_SET WEB DISPLAY LIMITS
- _o_SET WEB TIMEOUT
- _o_USE EXTERNAL DATABASE
- _o_USE INTERNAL DATABASE
- _o_Web Context
- _o_Win to Mac
- _o_XSLT APPLY TRANSFORMATION
- _o_XSLT GET ERROR
- _o_XSLT SET PARAMETER