4D v11 SQL

Language Reference
Windows® / Mac OS®

4D v11 SQL Language Reference
Version 11.1 for Windows® and Mac OS®

Contents

8. Communications..... 365

9. Compiler.....393

13. Debugging.....511

14. Drag and Drop..... 555

15. Entry Control..... 579

16. External Data Source.....603

17. Form Events.....625

27. Menus..... 907

33. Pasteboard..... 1081

34. Pictures..... 1105

35. Printing.....1143

36. Process (Communications). 1195

37. Process (User Interface)..... 1213

38. Processes..... 1219

39. Queries.... 1255

40. Quick Report....1293

42. Records....1373

43. Relations....1397

46. Selection..... 1483

47. Sets..... 1517

48. SQL.... 1543

49. String....1563

52. System Documents....1649

53. System Environment..... 1711

58. User Forms.....1799

59. User Interface.....1809

62. Web Server....1899

1

Introduction

4D has its own programming language. This built-in language, consisting of nearly 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 Developer 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 Developer 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.: OPEN 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.

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.

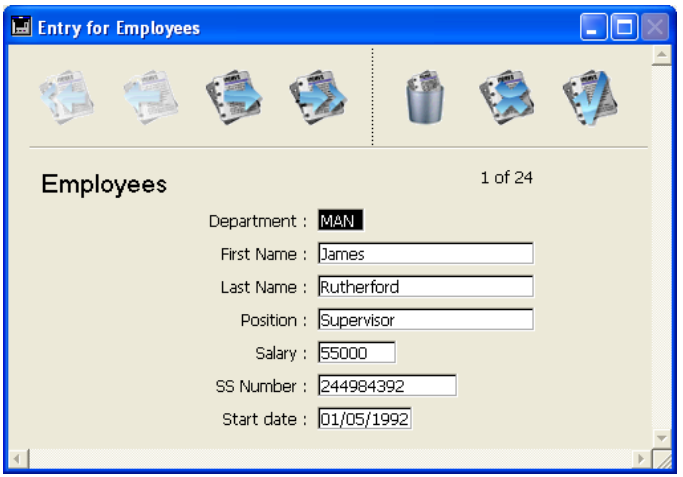
Getting started with object methods

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.

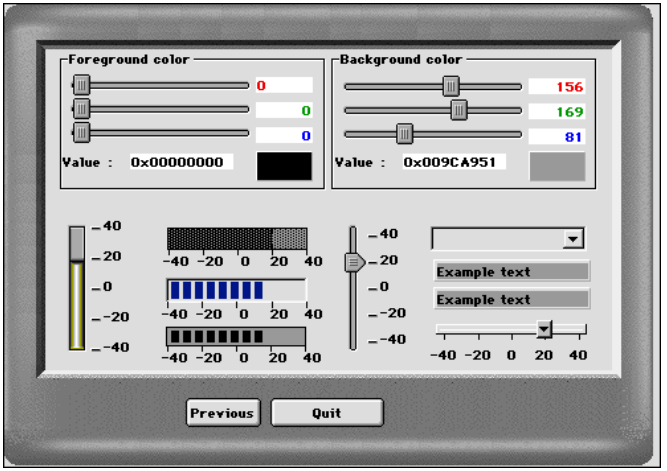
Object methods are the primary tools for managing the user interface, which is the doorway to your database. The user interface consists of the procedures and conventions by which a computer communicates with the user. The goal is to make the user interface of your database as simple and easy to use 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

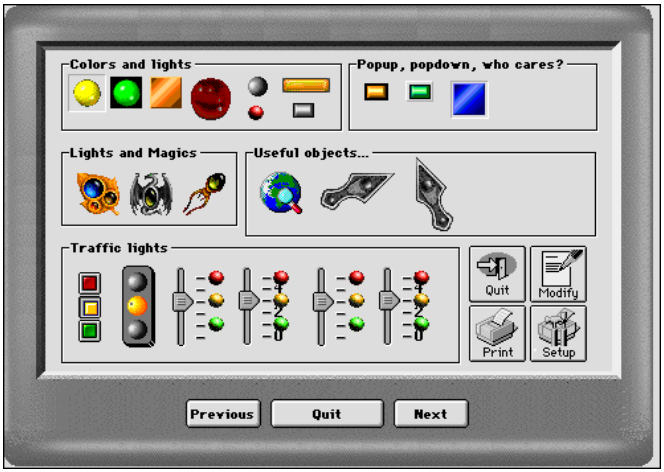
4D enables you to build classic forms, such as the one shown here:



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



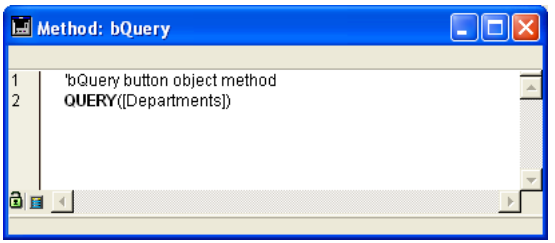
You can even build forms that incorporate a graphical flair limited only by your imagination:



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. You can even keep them in the Clipbook (Windows) or Scrapbook (Macintosh), ready to be used when you need them.

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, see the section Form event.

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 about Triggers, see the section Triggers.

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 section Database Methods.

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 section Building a 4D application.

• 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.

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

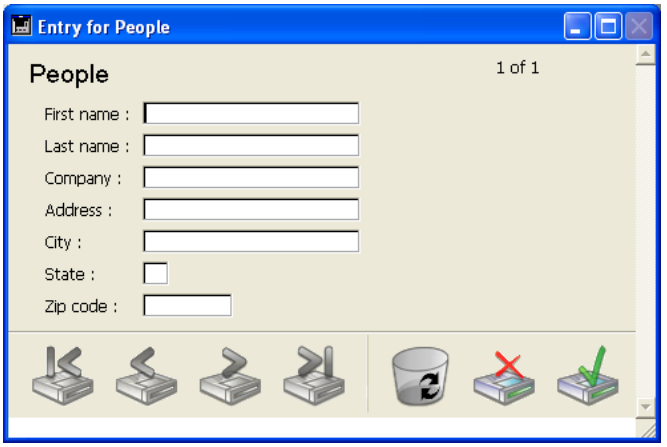
Custom menus are the primary interface in an application. They make it easier for users to learn and use a database. Creating custom menus is very simple—you associate methods or automatic actions with each menu command (also called menu items) in the Menu editor.

“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.

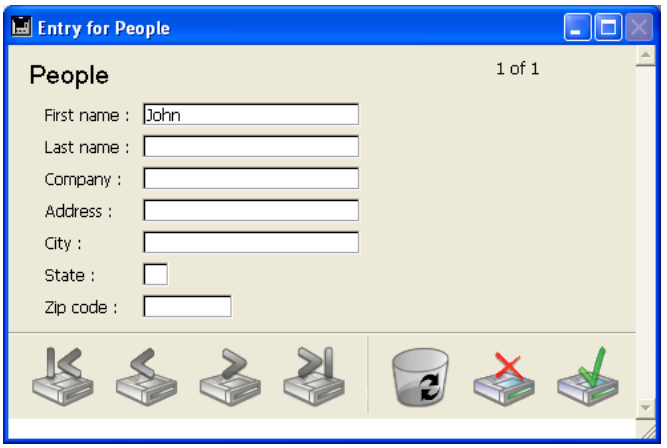
The User's Perspective

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

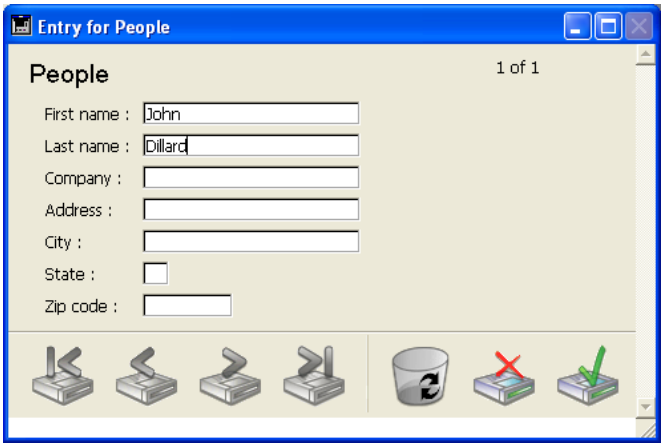
The Input form for the People table is displayed.



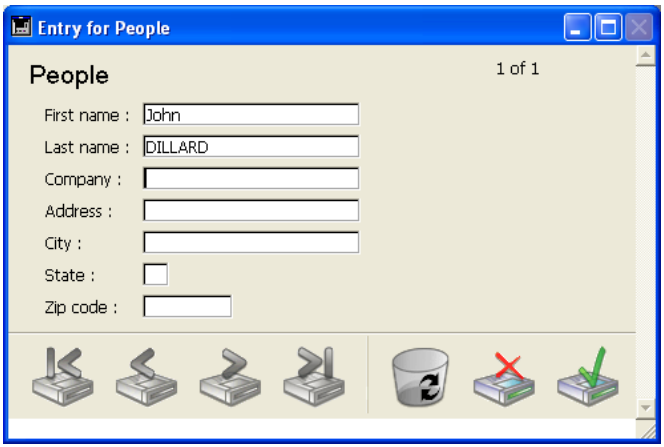
The user enters the person's first name and then tabs to the next field.



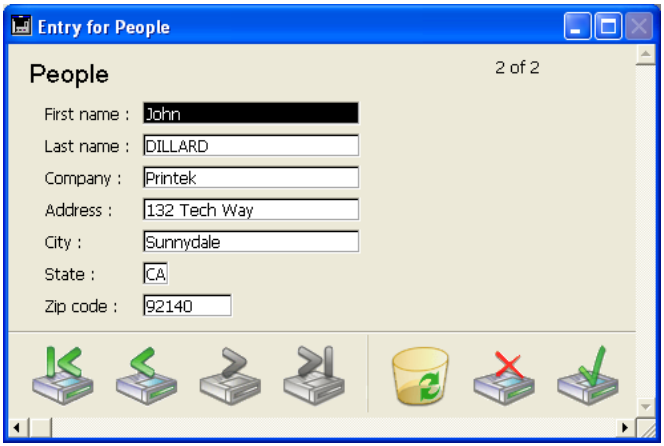
The user enters the person's last name.



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

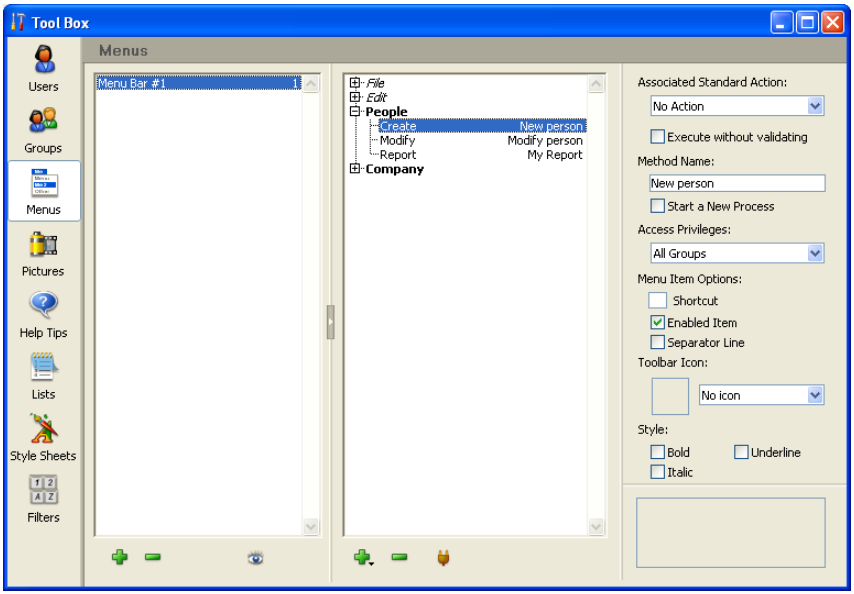


The user finishes entering the record and clicks the validation button (generally the last button in the button bar).

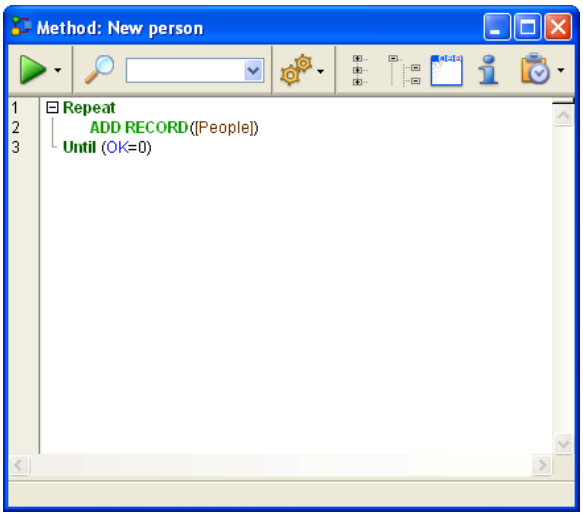


Another blank record appears, and the user clicks the Cancel button (the one with the “X”) to terminate the “data entry loop.” The user is returned to the menu bar.

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 Person associated with it. This method was created in the Design environment, using the Method editor.



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

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

```
Last Name:=Uppercase(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 Person 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

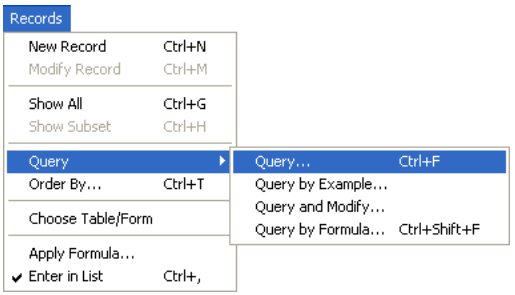
The next section, "Using a Database in the Design Environment," displays the tasks performed in the Design environment.

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

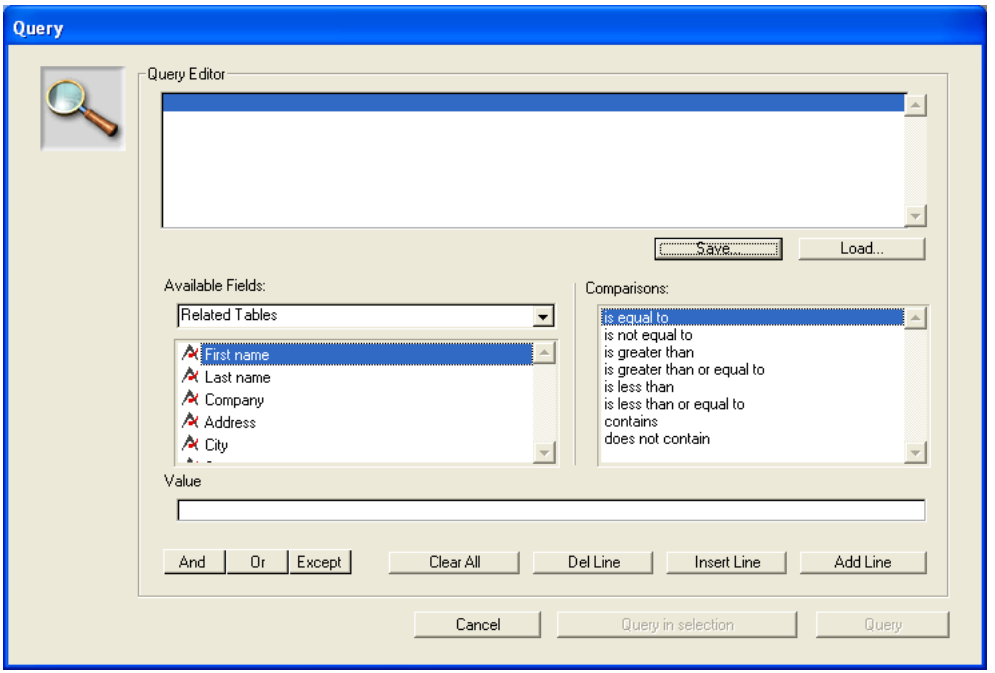
Note that although both methods perform the same task, the steps in the second section are automated using the language.

Using a database in the Design environment

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



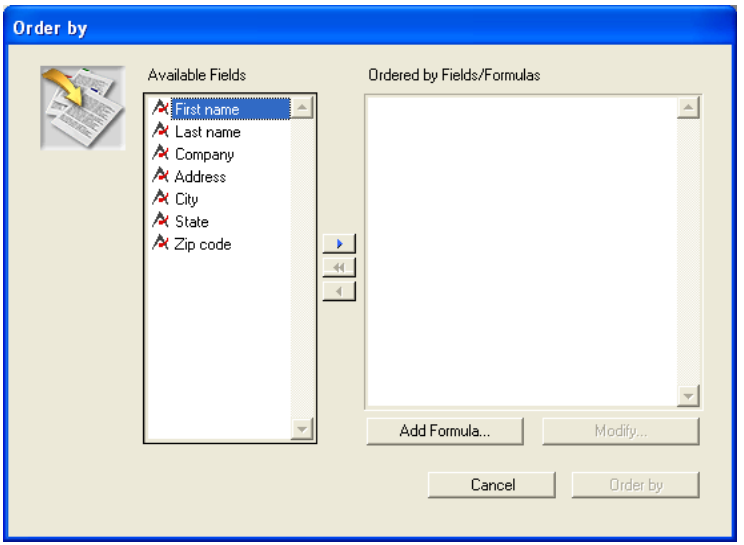
The **Query** editor is displayed.



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 **People** 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!

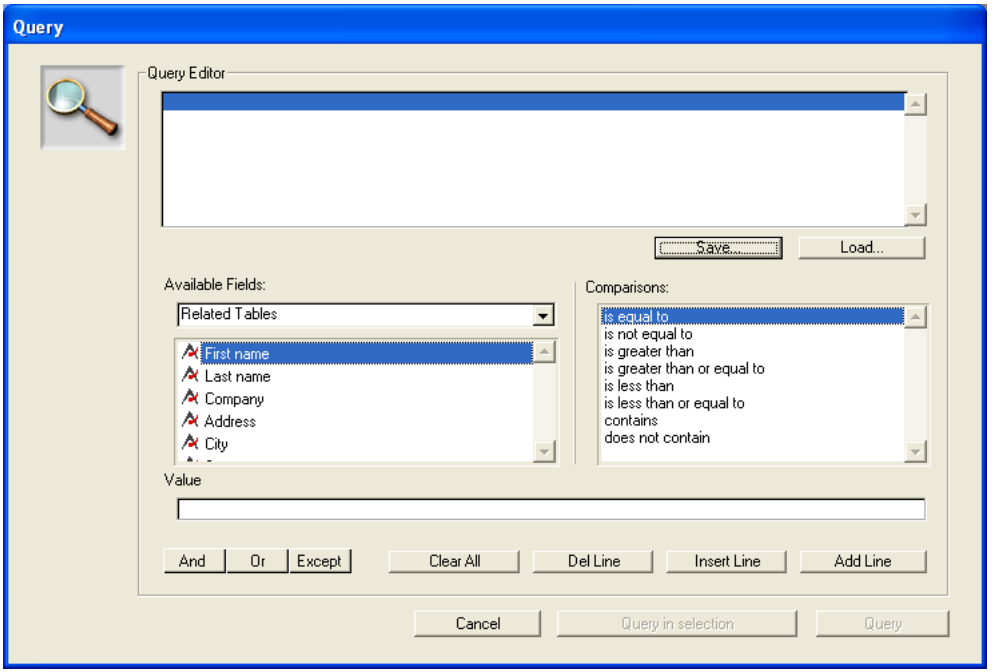
A method called My Report is attached to the menu command; it looks like this:

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

The first line is executed:

```
QUERY ([People])
```

The **Query** editor is displayed.



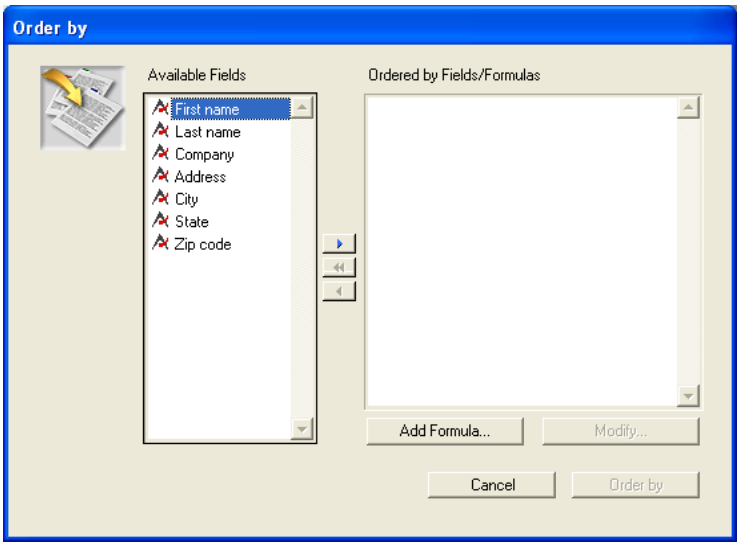
The user enters the criteria and clicks the **Query** button. The query is performed.

The second line of the My Report method is executed:

ORDER BY ([People])

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 My Report method is executed:

OUTPUT FORM ([People]; "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 My Report method is executed:

PRINT SELECTION ([People])

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

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 My Report method.

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

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

The first line is executed:

```
QUERY([People];[People]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 My Report2 method is executed:

```
ORDER BY([People];[People]Last Name;>;[People]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 My Report2 method are executed:

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

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.

4D plug-ins

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

- **4D Write**: Word-processor
- **4D Draw**: Graphical drawing program
- **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
- **4D Open for Java**: Connectivity with Java applications
- **4D Open for 4D**: Connectivity (from 4D to 4D) for building distributed 4D information systems.

For more information, contact 4D or its Partners. Visit our Web Sites:

USA & International:
http://www.4d.com

France:
http://www.4d.fr

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.

2

Language Definition

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.
• **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.

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 seven basic data types: string, numeric, date, time, Boolean, picture, and pointer.

• **String**: A series of characters, such as “Hello there”. Alpha and Text fields, and string and text variables, 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.

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	!1/1/1989! + 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 section 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 seven expression types:
• String expression
• Numeric expression (also referred to as number)
• Date expression
• Time expression
• Boolean expression
• Picture expression
• Pointer expression
The following table gives examples of each of the seven types of expressions.

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.
!1/25/97!	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 command “Current date” 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.

See Also

Arrays, Constants, Data Types, Methods, Operators, Pointers, Variables.

4D fields, variables, and expressions can be of the following data types:

Data Type	Field	Variable	Expression
String (see note 1)	Yes	Yes	Yes
Number (see note 2)	Yes	Yes	Yes
Date	Yes	Yes	Yes
Time	Yes	Yes	Yes
Boolean	Yes	Yes	Yes
Picture	Yes	Yes	Yes
Pointer	No	Yes	Yes
BLOB (see note 3)	Yes	Yes	No
Array (see note 4)	No	Yes	No
Integer 64 bits (see note 5)	Yes	No	No
Float (see note 5)	Yes	No	No
Undefined	No	Yes	Yes

Notes:

1. String includes alphanumeric field, fixed length variable, and text field or variable.
2. Number includes Real, Integer, and Long Integer field and variable.
3. BLOB is an acronym for Binary Large OBject. For more information about BLOBs, see the section BLOB Commands.
4. Array includes all types of arrays. For more information, see the section Arrays.
5. The Integer 64 bits and Float types are only managed via SQL. It is not recommended to work with them via the 4D language because in this case they are converted into the Real type which could lead to some loss of accuracy.

String

String is a generic term that stands for:
- Alphanumeric fields or variables
- Text fields or variables
- Any String or Text expression

A string is composed of characters. The handling of character strings varies depending on whether 4D is run in Unicode mode or in non-Unicode mode (compatibility mode). This mode is set via the application Preference (see the About Unicode section).

Unicode Mode
• An Alphanumeric field may contain from 0 to 255 characters (limit is set during field definition).
• A Text field, variable, or expression may contain from 0 to 2 GB of text.
• There is no difference between a string or text variable.

Non-Unicode Mode (compatibility)
Each character can be one of the 256 ASCII characters supported by Windows and Mac OS. For more information about ASCII codes, please refer to the ASCII Codes section.
• An alphanumeric field may contain from 0 to 255 characters (limit is set during field definition).
• A Fixed length variable may contain from 0 to 255 characters (limit depends on the variable declaration).
• A Text field, variable, or expression may contain from 0 to32,000 characters .

No matter what the mode, youcan assign a string to a text field and vice-versa; 4D does the conversion, truncating if necessary. You can mix string and text in an expression.

Note: In the *4D Language Reference* manual, both string and text parameters in command descriptions are denoted as String, except when marked otherwise.

Number

Number is a generic term that stands for:
• Real Field, variable or expression
• Integer field, variable or expression
• Long Integer field, variable or expression

The range for the Real data type is ±1.7e±308 (15 digits)
The range for the Integer data type (2-byte Integer) is -32,768..32,767 (2^15..(2^15)-1)
The range for the Long Integer data type (4-byte Integer) is -2^31..(2^31)-1

You can assign any Number data type to another; 4D does the conversion, truncating or rounding if necessary. However, when values are out of range, the conversion will not return a valid value. You can mix Number data types in expressions.

Note: In the *4D Language Reference* manual, no matter the actual data type, the Real, Integer, and Long Integer parameters in command descriptions are denoted as Number, except when marked otherwise.

Date

- A Date field, variable or expression can be in the range of 1/1/100 to 12/31/32,767.
- Using the US English version of 4D, a date is ordered month/day/year.
- If a year is given as two digits, it is assumed to be in the 1900's if the value is greater than or equal to 30, and the 2000's if the value is less than 30 (this default can be changed using the SET DEFAULT CENTURY command).

Note: In the *4D Language Reference* manual, Date parameters in command descriptions are denoted as Date, except when marked otherwise.

Time

- A Time field, variable or expression can be in the range of 00:00:00 to 596,000:00:00.
- Using the US English version of 4D, time is ordered hour:minute:second.
- Times are in 24-hour format.
- A time value can be treated as a number. The number returned from a time is the number of seconds that time represents. For more information, see the section Time Operators.

Note: In the *4D Language Reference* manual, Time parameters in command descriptions are denoted as Time, except when marked otherwise.

Boolean

A Boolean field, variable or expression can be either TRUE or FALSE.

Note: In the *4D Language Reference* manual, Boolean parameters in command descriptions are denoted as Boolean, except when marked otherwise.

Picture

A Picture field, variable or expression can be any Windows or Macintosh picture. In general, this includes any picture that can be put on the pasteboard or read from the disk using 4D or Plug-In commands.

Note: In the *4D Language Reference* manual, Picture parameters in command descriptions are denoted as Picture, except when marked otherwise.

Pointer

A Pointer variable or expression is a reference to another variable (including arrays and array elements), table, or field. There is no field of type Pointer.

For more information about Pointers, see the section Pointers.

Note: In the *4D Language Reference* manual, Pointer parameters in command descriptions are denoted as Pointer except when marked otherwise.

BLOB

A BLOB field or variable is a series of bytes (from 0 to 2 GB in length) that you can address individually or by using the BLOB Commands. There is no expression of type BLOB.

Note: In the *4D Language Reference* manual, BLOB parameters in command descriptions are denoted as BLOB.

Array

Array is not actually a data type. The various types of arrays (such as Integer Array, Text Array, and so on) are grouped under this title. Arrays are variables—there is no field of type Array, and there is no expression of type Array. For more information about arrays, see the section Arrays.

Note: In the *4D Language Reference* manual, Array parameters in command descriptions are denoted as Array, except when marked otherwise (i.e., String Array, Numeric Array, ...).

Undefined

Undefined is not actually a data type. It denotes a variable that has not yet been defined. A function (a project method that returns a result) can return an undefined value if, within the method, the function result ($0) is assigned an undefined expression (an expression calculated with at least one undefined variable). A field cannot be undefined.

Converting Data Types

The 4D language contains operators and commands to convert between data types, where such conversions are meaningful. The 4D language enforces data type checking. For example, you cannot write: "abc"+0.5+!12/25/96!-?00:30:45?. This will generate syntax errors.

The following table lists the basic data types, the data types to which they can be converted, and the commands used to do so:

Data Type	Convert to String	Convert to Number	Convert to Date	Convert to Time
String		Num	Date	Time
Number (*)	String			
Date	String			
Time	String			
Boolean		Num		

(*) Time values can be be treated as numbers.

Note: In addition to the data conversions listed inthis table, more sophisticated data conversions can be obtained by combining operators and other commands.

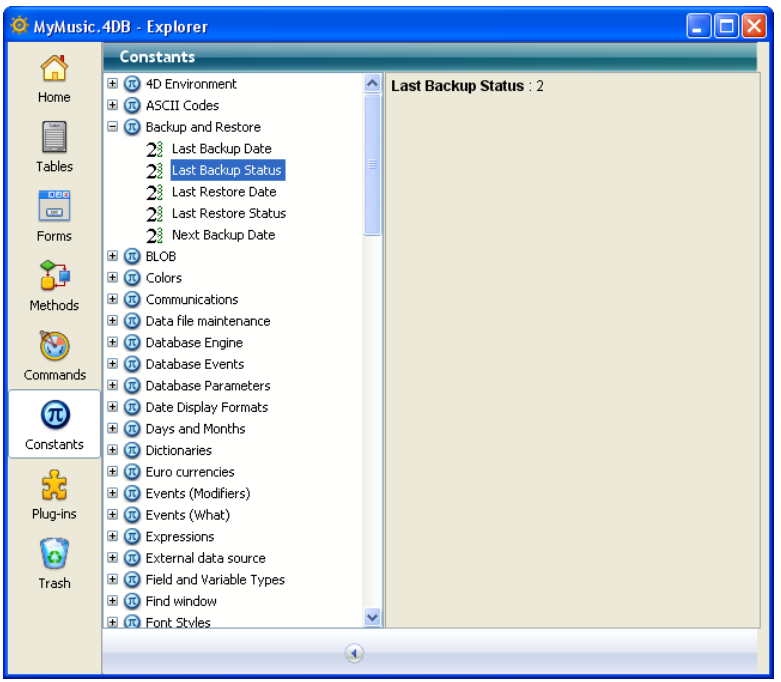
See Also

Arrays, Constants, Control Flow, Identifiers, Methods, Operators, Pointers, Type, Variables.

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

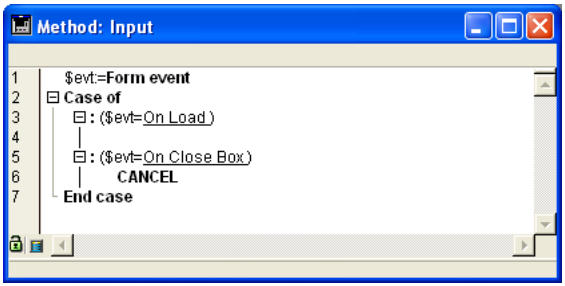
4D provides a set of **predefined constants.** These constants are grouped by themes in the Explorer Window:



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 type-ahead mechanism suggests constants that correspond to the programming context.

Predefined constant names can contain up to 31 characters.

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
```

Date Constants

A date constant is enclosed by exclamation marks (!…!). In the US English version of 4D, a date is ordered month/day/year, with a slash (/) setting off each part. Here are some examples of date constants:

```
!1/1/76!
!4/4/04!
!12/25/96!
```

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.

Note: A two-digit year is assumed to be in the 1900's. Unless this default setting has been changed using the command SET DEFAULT CENTURY.

Time Constants

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

Note: This syntax can be used on both Windows and Macintosh. On Macintosh, you can also use the Dagger symbol (Option-T on a US keyboard).

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.

See Also

Control Flow, Data Types, Identifiers, Methods, Operators, Pointers, 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 (15 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
- 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 section Compiler Commands.

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 section 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 with the same name. 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 section Form event.

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.

See Also

Arrays, Constants, Control Flow, Data Types, Identifiers, Methods, Operators, Pointers.

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.

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.

Document

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

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

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 can only be used in a method installed by the ON ERR CALL command. This variable contains the error code. 4D error codes and system error codes are listed in the Error Codes section.

MouseDown, MouseX, MouseY, KeyCode, Modifiers and MouseProc

These system variables can only be used in a method installed by the ON EVENT CALL command.

- 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.

Note: When a picture field or variable is clicked, the MouseX and MouseY system variables return the local coordinates of the click in the On Clicked or On Double clicked form event. For more information, please refer to the Pictures section.

- 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 ASCII Codes 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.

See Also

Sets, Variables.

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…
    ENABLE BUTTON (MyButton) ` enable the button
Else ` Otherwise…
    DISABLE BUTTON (MyButton) ` 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…
    ENABLE BUTTON($1->) ` enable the button
Else ` Otherwise…
    DISABLE BUTTON($1->) ` 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:

```
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.

See Also

Arrays, Arrays and Pointers, Constants, Control Flow, Data Types, Identifiers, Methods, Operators, Variables.

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.
- 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])
INPUT FORM ([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.

Do not start a field name with the underscore character (_). The underscore character is reserved for plug-ins. When 4D encounters this character at the beginning of a field in the Method editor, it removes the underscore.

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

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

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"
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.

See Also

Arrays, Constants, Data Types, Methods, Operators, Pointers, Variables.

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.

See Also

Logical Operators, Methods.

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

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

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

```
If (Boolean_Expression)
    statements(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.

Example

```
    ` Ask the user to enter the 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)
    statements(s)
Else
End if
```

See Also

Case of...Else...End case, Control Flow, For...End for, Repeat...Until, While...End while.

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
```

See Also

Control Flow, For...End for, If...Else...End if, Repeat...Until, 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 section 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.

See Also

Case of...Else...End case, Control Flow, For...End for, If...Else...End if, 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)
```

See Also

Case of...Else...End case, Control Flow, For...End for, If...Else...End if, While...End while.

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]])=Char(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]])=Char(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 for
    End for
```

See Also

Case of...Else...End case, Control Flow, If...Else...End if, Repeat...Until, While...End while.

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 section Form event.

• **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, 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, 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 moreinformation about D atabase Methods, see the section Database Methods.

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 section Form event.
- Triggers, see the section Triggers.
- Project methods, see the section Project Methods.
- Database methods, see the section Database Methods.

See Also

Arrays, Constants, Control Flow, Data Types, Database Methods, Identifiers, Operators, Pointers, Triggers, Variables.

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 section 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
INPUT FORM([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.

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
    ` ...
    ` Let's say [People]Last Name is equal to "williams"
[People]Last Name:=DO SOMETHING ([People]Last Name)
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).

See Also

Control Flow, Database Methods, Methods.

3

4D Environment

ADD DATA SEGMENT

Parameter	Type	Description

This command does not require any parameters

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.

Application file → String

Parameter	Type		Description
This command does not require any parameters			
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 Developer located at \PROGRAMS\4D on the volume E, the command returns E:\PROGRAMS\4D\4D Developer.EXE.

On Macintosh
If, for example, you are running 4D Developer in the Programs folder on the disk Macintosh HD, the command returns Macintosh HD:Programs:4D Developer.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 section System Documents.

See Also

Data file, DATA SEGMENT LIST, Structure file.

Application type

Application type → Long Integer

Parameter	Type	Description
This command does not require any parameters		
Function result	Long Integer←	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 Developer	Long Integer	0
4D Unlimited Desktop	Long Integer	1
4D Interpreted Desktop	Long Integer	2
4D Desktop	Long Integer	3
4D Client	Long Integer	4
4D Server	Long Integer	5
4D First	Long Integer	6

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
```

See Also

Application version, Version type.

Application version {(*)} → String

Parameter	Type		Description
*	*	→	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	Update number
4	Revision number

Example: The string "0600" stands for version 6.0.0.

• 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	Update number
8	Revision number

Example: The string "B0120602" would stand for the Beta 12 of version 6.0.2.

Examples

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]])
```

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
```

See Also

Application type, Version type.

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 *Design Reference* manual of 4D).
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.

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 (“.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* 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 (structurePath; dataPath{; archiveFolder{; option{; method}}})

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	Number	→	Compacting options
method	Text	→	Name of 4D callback method

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) YYYYMM-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 Developer 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 data 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:
- Do not create log file (16384): Generally, this command creates a log file in XML format. Its name is based on that of the data file and it is placed next to this file (original location). For example, for a data file named "data.4dd," the log file will be named "data_compact_log.xml." With this option, no log file will be created.
- Create process (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.

• 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). Its name is based on that of the data file and it is placed next to this file. For example, for a data file named "data.4dd," the log file will be named "data_compact_log.xml."

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)
```

See Also

VERIFY DATA FILE.

System Variables or Sets

If the compacting operation is carried out correctly, the OK system variable is set to 1; otherwise, it is set to 0.

COMPONENT LIST (componentsArray)

Parameter	Type		Description
componentsArray	Array string	←	Names of the components

Description

When a database is opened, 4D loads the valid components found in the Components folder that is next to the structure file. 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.

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 (accessPath)

Parameter	Type		Description
accessPath	String	→	Name or complete access path of the data file to create

Description

The CREATE DATA FILE command allows creating a new data file to disk and to replace 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: This command cannot be used with 4D Client or 4D Server.

See Also

OPEN DATA FILE.

Data file {(segment)} → String

Parameter	Type		Description
segment	Number	→	Segment number
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 or one data segment for the database with which you are currently working.

If you do not pass the segment parameter, it returns the long name of the data file or the first segment (if the database is segmented). If you pass the segment parameter, it returns the long name of the corresponding data segment. If you pass a segment number greater than the number of data segments, it returns an empty string.

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*f*: on the disk Macintosh HD, a call to Data file returns Macintosh HD:Documents:MyCDs*f*: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 while running 4D Client, only the name of the data file or the first data segment is returned, not the long name. In addition, even though the database is segmented, the command returns an empty string for the other data segments. If you need (for adminstrative purposes) to display a list of the data segments on a 4D Client station, use a Stored Procedure to build the data segment list and store it in a variable on the server machine, then get the contents of this variable using the GET PROCESS VARIABLE command.

Example

The following code goes through the data segments of a database.

```
If (Application type#4D Client)
    $vlDataSegNum:=0
    Repeat
        $vlDataSegNum:=$vlDataSegNum+1
        $vsDataSegName:=Data file($vlDataSegNum)
        If ($vsDataSegName#"")
            ALERT ("Data segment "+String($vlDataSegNum)+":"+Char(34)+
                                                    $vsDataSegName+Char(34)+".")
        End if
    Until ($vsDataSegName="")
    ALERT("There is/are "+String($vlDataSegNum-1)+"data segment(s).")
End if
```

See Also

Application file, DATA SEGMENT LIST, Structure file.

DATA SEGMENT LIST (segments)

Parameter	Type	Description
segments	String array ←	Long names of data segments for the database

Description

DATA SEGMENT LIST populates the segments array with the long names of the data segments for the database with which you are currently working.

Compatibility Note: Starting with version 11 of 4D, data segments are no longer supported (the size of the data file is now unlimited). This command has been kept for compatibility reasons. It now systematically returns an array with one element containing the long name of the data file of the database.

See Also

Application file, Data file, Structure file.

FLUSH BUFFERS

Parameter	Type	Description

This command does not require any parameters

Description

The FLUSH BUFFERS command immediately saves the data buffers to disk. All changes that have been made to the database are stored on disk.

You usually do not need to call this command, as 4D saves data modification on a regular basis. The **Flush Data Buffers every X Minutes** option (Data Management page of the Preferences), which specifies how often to save, is typically used to control buffer flushing.

Note: 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.

Get 4D folder ({folder}{; }{*}) → String

Parameter	Type		Description
folder	Longint	→	Folder type (if omitted = active 4D folder)
*	*	→	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
Active 4D Folder	Longint	0 (default)
Licenses Folder	Longint	1
Extras Folder	Longint	2
4D Client Database Folder	Longint	3
Database Folder	Longint	4
Database Folder Unix Syntax	Longint	5
Current Resources folder	Longint	6

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 the **4D** folder to store the following information:

- User registration files
- Preferences files used by the 4D environment applications, tools, and utility programs
- TCP/IP Network protocol option file
- Local database folders created by 4D Client for storing elements downloaded from 4D Server (resources, plug-ins, Extras folder, etc.).

The 4D folder is created by default at the following location:

- On Windows: {Disk}:\Documents and Settings\All Users\Application Data\4D
- On Mac OS: {Disk}:Library:Application Support:4D

With the 4D Client application or if the All Users folder is locked, the active 4D folder is created at the following location:

- Under Windows: {Disk}:\Documents and Settings\User\Application Data\4D
- Under Mac OS: {Disk}:Users:User:Library:Application Support:4D

Licenses Folder
Folder containing the Licenses files of the machine.
The **Licenses** folder is placed at the following location:

- On Windows: {Disk}:\Documents and Settings\All Users\Application Data\4D\Licenses
- On Mac OS : {Disk}:Library:Application Support:4D:Licenses

Extras Folder
Folder with customized contents downloaded to each 4D Client machine.
You use this folder for transferring custom items from the server to the client machines (resources file, text documents, XML preferences files, etc.). The original hierarchy of the folder is reconstructed on each client machine.
4D Server automatically manages the modifications made to this folder and only transfers what is necessary. Moreover, the contents of the folder is compressed in order to optimize network copying time.
On the 4D Server or 4D Developer side, the original Extras folder should be placed next to the database structure file.
On the 4D Client side, the **Extras** folder is downloaded to the following location on each client machine, i.e.:

- On Windows: {Disk}:\Documents and Settings\Current user\Application Data\4D\DatabaseName_Address\Extras
- On Mac OS: {Disk}:Users:User:Library:Application Support:4D:DatabaseName_Address:Extras

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, Extras folder, etc.).
The **4D Client Database Folder** is placed at the following location on each client machine:
• On Windows: {Disk}:\Documents and Settings\Current user\Application Data\4D\DatabaseName_Address
• On Mac OS: {Disk}:Users:User:Library:Application Support: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 Mac OS or the SET CGI EXECUTABLE command.

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 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.

Examples

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
```

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).

See Also

COMPONENT LIST, System folder, Temporary folder, Test path name.

Get current database localization

Get current database localization → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Current language of the database

Description

The Get current database localization command returns the current language used by the database, 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 the *Design Reference* manual.

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 parameter ({aTable; }selector{; stringValue}) → Longint

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	Longint	←	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
Seq Order Ratio	**** *Selector disabled* ****	
Seq Access Optimization	**** *Selector disabled* ****	
Seq Distinct Values Ratio	**** *Selector disabled* ****	
Index Compacting	**** *Selector disabled* ****	
Seq Query Select Ratio	**** *Selector disabled* ****	
Minimum Web Process	Longint	6
Maximum Web Process	Longint	7
Web conversion mode	Longint	8
Database Cache Size	Longint	9
4D Developer Scheduler	Longint	10
4D Server Scheduler	Longint	11
4D Client Scheduler	Longint	12
4D Server Timeout	Longint	13
4D Client Timeout	Longint	14
Port ID	Longint	15
IP Address to listen	Longint	16

Character set	Longint	17
Max Concurrent Web Processes	Longint	18
Client Minimum process Web	Longint	19
Client Maximum process Web	Longint	20
Client Maximum Web requests size	Longint	21
Client Port ID	Longint	22
Client IP Address to listen	Longint	23
Client Character set	Longint	24
Client Max Concurrent Web Proc	Longint	25
Cache Writing Mode	**** *Selector disabled* ****	
Maximum Web requests size	Longint	27
4D Server Log Recording	Longint	28
Web Log Recording	Longint	29
Client Web Log Recording	Longint	30
Table Sequence Number	Longint	31
Real Display Precision	Longint	32
TCP_NODELAY	Longint	33
Debug Log Recording	Longint	34
Client Server Port ID	Longint	35
WEDD Signature	Longint	36
Invert Objects	Longint	37
HTTPS Port ID	Longint	39
Client HTTPS Port ID	Longint	40
Unicode mode	Longint	41
Temporary memory size	Longint	42
SQL Autocommit	Longint	43
SQL Engine Case Sensitivity	Longint	44

To know the values that could be returned by this function, as well as the scope of each selector and whether or not any changes made are kept between sessions, please refer to the description of the SET DATABASE PARAMETER command.

The Database Cache Size (9) selector allows you to get the current database memory cache size. The returned value is expressed in bytes.
The Maximum Cache size is set on the “Database/Data Management” page of the Preferences. The actual size allocated to the database cache however will depend on both the settings and the current system resources. The Get database parameter command allows you to get the actual size of the memory allocated to the database cache by 4D.

Note: You cannot set the database cache memory size using the language. In other words, the Database Cache Size selector cannot be set using the SET DATABASE PARAMETER command.

When you use the WEDD Signature (36) selector with this command, the string defined as the WEDD signature is returned in the optional stringValue parameter and the function returns 0.

Examples

1. The following method allows you to get 4D scheduler current values:

```
C_LONGINT($ticksbtwcalls;$maxticks;$minticks;$lparams)
If (Application type=4D Developer) ` 4D Developer is used
    $lparams:=Get database parameter(4D Developer scheduler)
    $ticksbtwcalls:=$lparams & 0x00ff
    $maxticks:=($lparams>>8) & 0x00ff
    $minticks:=($lparams>>16) & 0x00ff
End if
```

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
```

See Also

DISTINCT VALUES, QUERY SELECTION, SET DATABASE PARAMETER.

GET SERIAL INFORMATION

version 6.7

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 Developer, 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*.

See Also

Get component resource ID.

Is compiled mode

Is compiled mode {(*)} → Boolean

Parameter	Type		Description
*	*	→	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
   ` ...
```

See Also

IDLE, Undefined.

Is data file locked → Boolean

Parameter	Type		Description
This command does not require any parameters			
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
```

OPEN 4D PREFERENCES (selector)

Parameter	Type		Description
selector	String	→	Key designating a theme or a page or a group of parameters in the Preferences dialog box

Description

The OPEN 4D PREFERENCES command provokes the display of the Preferences dialog box of the current 4D application and the display of the theme or page corresponding to the key passed in selector.

The selector parameter must contain one or more “keys” indicating a theme, page or group of parameters in the Preferences dialog box. The list of keys that can be used is provided below.

You can pass either a fixed access path or the name of a single element in selector:
• **Fixed access path**: The selector parameter is put together in the following manner: /Theme{/Page{/Parameter group}}.
The string must start with the / character and each level must be separated with a /.
For example, to set the Compiler page of the Design Mode theme, selector must contain "/Design Mode/Compiler".
• **Name** (relative path): In this case, the selector parameter cannot start with the / character. Simply pass the name of the desired element and 4D will open the first corresponding element in the following search order: parameter group -> page-> theme.
For example, if you pass “Progress Indicator” in selector, 4D will open the Options page of the Application theme.

To open the dialog box directly on the first page, simply pass “/” in selector.

The command opens the Preferences page on the element specified in selector; however, all other themes and pages remain accessible. It is up to the developer to make sure that user access to Preferences does not hinder the application. To control user actions, it is recommended that you enable the user access management system.

Path keys

The following is a list of keys that can be used in the selector parameter:

/Application
/Application/Options
/Application/Options/Options
/Application/Options/Temporary Folder Location
/Application/Options/Drag and Drop Highlight
/Application/Options/Progress Indicator
/Application/Options/Display Toolbar
/Application/Options/Display Windows
/Application/Access
/Application/Access/Data Access
/Application/Access/General Settings
/Application/Access/User Access
/Application/CPU Priorities
/Application/CPU Priorities/Set CPU Priority to:
/Application/Shortcuts
/Application/Shortcuts/Keys
/Application/Compatibility
/Application/Compatibility/Design Compatibility
/Application/Compatibility/Web Compatibility
/Application/Compatibility/Platform
/Design Mode
/Design Mode/Structure
/Design Mode/Structure/General Font
/Design Mode/Structure/Forms and Methods Automatic Comments
/Design Mode/Structure/Automatic Form Creation
/Design Mode/Form Editor
/Design Mode/Form Editor/Object Templates
/Design Mode/Form Editor/Move
/Design Mode/Form Editor/Auto Alignment
/Design Mode/Form Editor/Default Display
/Design Mode/Method Editor
/Design Mode/Method Editor/Font
/Design Mode/Method Editor/Default Display
/Design Mode/Method Editor/Options
/Design Mode/Method Editor/Syntax
/Design Mode/Compiler
/Design Mode/Compiler/Compilation Options
/Design Mode/Compiler/Compiler Methods for...
/Design Mode/Documentation

/Design Mode/Moving
/Design Mode/Moving/Default Actions during the Copy if Dependent Objects
/Design Mode/Moving/Moving Dialog
/Design Mode/Documentation/Documentation Access from the Explorer
/Database
/Database/Data Management
/Database/Data Management/General
/Database/Data Management/Database Cache Settings
/Database/Data Management/WEDD
/Database/International
/Database/International/Script Manager
/Database/International/Right-to-left Languages
/Database/International/Numeric Display Format
/Backup
/Backup/Configuration
/Backup/Configuration/Backup Contents
/Backup/Configuration/Backup File Destination Folder
/Backup/Configuration/Last Backup Information
/Backup/Configuration/Log Management
/Backup/Scheduler
/Backup/Scheduler/Backup Frequency
/Backup/Backup
/Backup/Backup/General
/Backup/Backup/Archive
/Backup/Restore
/Backup/Restore/Automatic Restore
/Client-Server
/Client-Server/Configuration
/Client-Server/Configuration/Network
/Client-Server/Configuration/Client-Server Connections Timeout
/Client-Server/Configuration/Client-Server Communication
/Client-Server/Configuration/4D Open
/Client-Server/Publishing
/Client-Server/Publishing/Publishing Information
/Client-Server/Publishing/Allow-Deny Table Configuration
/Client-Server/Publishing/Encryption
/Web
/Web/Configuration
/Web/Configuration/Web Server Publishing
/Web/Configuration/Default HTML Path
/Web/Configuration/Starting Mode

/Web/Advanced
/Web/Advanced/Cache
/Web/Advanced/Web Process
/Web/Advanced/Options
/Web/Advanced/Web Passwords
/Web/Options
/Web/Options/Options
/Web/Options/Text Conversion
/Web/Options/Persisent Connections
/Web/Log Format
/Web/Log Format/Web Log Type
/Web/Log Format/Web Log Token Selection
/Web/Log Scheduler
/Web/Log Scheduler/Backup Frequency for Web Log File
/Web Services
/Web Services/SOAP
/Web Services/SOAP/Server Side
/Web Services/SOAP/Client Side
/SQL
/SQL/Configuration
/SQL/Configuration/SQL Server Access

Examples

1. Open Preferences on the first page:

```
OPEN 4D PREFERENCES("/")
```

2. Open the “Shortcuts” page of the “Application” theme:

```
OPEN 4D PREFERENCES("/Application/Shortcuts")
```

3. Open the “Advanced” page of the “Web” theme:

```
OPEN 4D PREFERENCES("Web Passwords")
```

System Variables or Sets

If the requested element is found and opened correctly, the system variable OK returns 1. Otherwise, it returns 0.

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 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. 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 is not possible 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, etc.

Before launching the operation, the command checks the validity of the specified data file: it must have the ".4dd" extension under Windows or have the "dat5" type under Mac OS. 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: This command cannot be used with 4D Client or 4D Server.

See Also

CREATE DATA FILE.

OPEN SECURITY CENTER

Parameter	Type	Description
This command 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.

See Also

VERIFY CURRENT DATA FILE.

PLUGIN LIST (numbersArray; namesArray)

Parameter	Type		Description
numbersArray	Longint Array	←	Numbers of plug-ins
namesArray	Array string	←	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.

See Also

COMPONENT LIST, Get plugin access, Is license available, SET PLUGIN ACCESS.

QUIT 4D {(time)}

Parameter	Type		Description
time	Number	→	Time (mn) 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 Developer/4D Client or on 4D Server.

With 4D Developer and 4D Client:
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 Developer or 4D Client.

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 minutes in time. This parameter is only taken into consideration during an execution on the server machine. It is ignored in 4D Client or 4D Developer.
If you do not pass a parameter for time, 4D Server will wait until all client machines are disconnected before quitting.

Unlike 4D Developer and 4D Client, 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 Stop 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
```

See Also
On Exit Database Method, On Server Shutdown Database Method.

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	Longint | 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 for selector	Value	Scope	Changes kept
Seq Order Ratio	1	**** *Selector disabled***	-
Seq Access Optimization	2	**** *Selector disabled***	-
Seq Distinct Values Ratio	3	**** *Selector disabled***	-
Index Compacting	4	**** *Selector disabled***	-
Seq Query Select Ratio	5	**** *Selector disabled* ****	-
Minimum Web Process	6	4D Developer, 4D Server (*)	Yes
Maximum Web Process	7	4D Developer 4D Server (*)	Yes
Web conversion mode	8	Current process	-
Database cache size	9	4D application (*) (**)	-
4D Developer Scheduler	10	4D application (*)	Yes
4D Server Scheduler	11	4D application (*)	Yes
4D Client Scheduler	12	4D application (*)	Yes
4D Server Timeout	13	4D application if positive value (***)	Yes
4D Client Timeout	14	4D application if positive value (***)	Yes
Port ID	15	4D Developer, 4D Server (*)	-
IP Address to listen	16	4D Developer, 4D Server (*)	Yes
Character set	17	4D Developer, 4D Server (*)	Yes
Max Concurrent Web Processes	18	4D Developer, 4D Server (*)	Yes

Client Minimum process Web	19	All 4D Client machines (*)	Yes
Client Maximum process Web	20	All 4D Client machines (*)	Yes
Client Max Web requests size	21	All 4D Client machines (*)	Yes
Client Port ID	22	All 4D Client machines (*)	Yes
Client IP Address to listen	23	All 4D Client machines (*)	Yes
Client Character set	24	All 4D Client machines (*)	Yes
Client Max Concurrent Web Proc	25	All 4D Client machines (*)	Yes
Cache writing mode	26	****Selector disabled ****	-
Maximum Web requests size	27	4D Developer, 4D Server (*)	Yes
4D Server Log Recording	28	4D Server, 4D Client (*)	-
Web Log Recording	29	4D Developer, 4D Server (*)	Yes
Client Web Log Recording	30	All 4D Client machines (*)	Yes
Table Sequence Number	31	4D Application	Yes
Real Display Precision	32	4D Application	-
TCP_NODELAY	33	4D Application (*)	-
Debug Log Recording	34	4D Application (*)	-
Client Server Port ID	35	Database (*)	Yes
WEDD Signature	36	Database (*)	Yes
Invert Objects	37	Database (*)	Yes
HTTPS Port ID	39	4D Developer, 4D Server (*)	Yes
Client HTTPS Port ID	40	All 4D Client machines (*)	Yes
Unicode mode	41	Database (*)	Yes
Temporary memory size	42	4D Application (*)	-
SQL Autocommit	43	Database (*)	Yes
SQL Engine Case Sensitivity	44	Database (*)	Yes

(*) The table parameter is ignored in this case.
(**) This selector can only be read (see Get database parameter command).
(***) If the value parameter is negative, the setting is local to the current process and is reset for the next request.

The value designates the value of the parameter. The value passed depends on the parameter that you want to modify. Here are the possible values for each selector:

Selector = 1 (Seq Order Ratio)
This selector has been disabled since it corresponds to a mechanism that has been optimized in 4D version 11.

Selector = 2 (Seq Access Optimization)
This selector has been disabled since it corresponds to a mechanism that has been optimized in 4D version 11.

Selector = 3 (Seq Distinct Values Ratio)
This selector has been disabled since it corresponds to a mechanism that has been optimized in 4D version 11.

Selector = 4 (Index Compacting)
This selector has been disabled since it corresponds to a mechanism that has been optimized in 4D version 11.

Selector = 5 (Seq Query Select Ratio)
This selector has been disabled since it corresponds to a mechanism that has been optimized in 4D version 11.

Selector = 6 (Minimum Web Process)
• Values: 0 -> 32,767
• Description: Minimum number of Web processes to maintain in non-contextual mode with 4D Developer and 4D Server. By default, the value is 0 (see below).

Selector = 7 (Maximum Web Process)
• Values: 0 -> 32,767
• Description: Maximum number of Web processes to maintain in non-contextual mode with 4D Developer and 4D Server. By default, the value is 10.
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).
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.

Selector = 8 (Web conversion mode)
• Values: 0, 1, 2 or 3
0= (Default mode) Conversion to the HTML 4.0 format if it is allowed by the browser. Otherwise, HTML 3.2 format + array use.
1= 6.0.x conversion mode
2= 6.5 conversion mode
3= Conversion to the HTML 4.0 format + CSS-P (since version 6.5.2)

• Description: Conversion mode of 4D forms for the Web with 4D Developer and 4D Server. By default, the 4D Web Server uses the cascading style sheets (CSS1) to generate HTML pages similar to the 4D forms displayed in 4D. With this feature, the forms might not convert correctly for databases created with versions of 4D prior to 6.7. Consequently, you have the possibility of setting the form conversion mode.
This mode is set only for the process (Web context) within which the SET DATABASE PARAMETER was called. It can be called from within the On Web Connection Database Method to ensure the compatibility of all the forms of a database, or just before a single form is displayed. If the command is called outside either the contextual mode or a Web process, it has no effect.

Note: An additional selector can be used with the Get database parameter command: Database Cache Size (9). This selector cannot be used with the SET DATABASE PARAMETER command. For more information, please refer to the description of the Get database parameter command.

Selector = 10 (4D Developer Scheduler)
Selector = 11 (4D Server Scheduler)
Selector = 12 (4D Client Scheduler)
• Values: For these three selectors, the value parameter is expressed in hexadecimal 0x00aabbcc detailed as follows:
aa = minimum number of ticks per call to the system (0 to 100 included).
bb = maximum number of ticks per call to the system (0 to 100 included).
cc = number of ticks between calls to the system (0 to 20 included).
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:
value = -1: maximum priority allocated to 4D,
value = -2: average priority allocated to 4D,
value = -3: minimum priority allocated to 4D.
• Description: This parameter allows you to dynamically set the 4D system internal calls. Depending on the selector, the scheduler value will be set for:
- 4D Developer (single-user applications), when the command is called from 4D Developer (selector=10).
- 4D Server when the command is called from 4D Server (selector=11).
- 4D Client when the command is called from 4D Client (selector=12).

Note: The operation of selector 12 (4D Client Scheduler) differs according to whether the SET DATABASE PARAMETER command is executed on the server machine or on the client machine:
- 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.
- 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.

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.
This operation is in effect in 4D starting with versions 6.8.6, 2003.3 and 2004.

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.

Selector = 13 (4D Server Timeout)
• Description: This parameter allows changing the value of the 4D Server timeout. The default 4D Server timeout value is defined on the "Client-Server/Configuration" page of the Preferences dialog box on the server side.
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:
- 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).
- 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.
To set the "No timeout" option, pass 0 in value.
(See example 1).

Selector = 14 (4D Client Timeout)
• Description: This parameter allows changing the value of the 4D Client timeout. The default 4D Client timeout value is defined on the "Client-Server/Configuration" page of the Preferences dialog box on the client side.
For more information about this selector, refer to 4D Server Timeout selector description (13).
The 4D Client Timeout selector can be used in very specific cases.

Selector = 15 (Port ID)
• Description: This parameter allows changing on the fly the TCP port ID used by the 4D Web server with 4D Developer 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.

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 Services, Configuration section.

Selector = 16 (IP Address to listen)
• Description: This parameter allows the user to change on the fly the IP address on which the 4D Web server will receive HTTP requests with 4D Developer and 4D Server. By default, no specific address is defined (value = 0). This parameter can be set in the Preferences of the database.
The 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).
You will pass in the value parameter a hexadecimal IP address. That is, to designate a IP address such as "a.b.c.d", you should write:

```
C_LONGINT($addr)
$addr:=($a<<24)|($b<<16)|($c<<8)|$d
SET DATABASE PARAMETER(IP Address to listen;$addr)
```

See also example 2. For more information on how to designate the IP address, refer to the Web Services, Web Server Settings section.

Selector = 17 (Character set)
• Values: The possible values depend on the operating mode of the database relating to the character set.
Unicode Mode : When the application is operating in Unicode mode, the values to pass for this parameter are character set identifiers (*MIBEnum*, 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:
4 = ISO-8859-1
12 = ISO-8859-9
13 = ISO-8859-10
17 = ShiftJIS
2026 = Big5
38 = euc-kr
106 = UTF-8
2250 = Windows-1250
2251 = Windows-1251
2253 = Windows-1253
2255 = Windows-1255
2256 = Windows-1256

Note: In the context of the Character set parameter, the Get database parameter command returns the IANA name of the character set in the optional stringValue parameter.

ASCII compatibility mode:
0: Western European
1: Japanese
2: Chinese
3: Korean
4: User-defined
5: Reserved
6: Central European
7: Cyrillic
8: Arabic
9: Greek
10: Hebrew
11: Turkish
12: Baltic

Note: For more information about Unicode mode, please refer to the description of selector 41.

• Description: This parameter allows the user to change on the fly the character set that the 4D Web Server (with 4D Developer 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 can be set in Preferences of the database. The Character set selector is useful for 4D Web Servers compiled and merged with 4D Desktop (in which there is no access to the Design mode).

Selector = 18 (Max Concurrent Web Processes)
• Values: You can pass any value between 10 and 32 000. The default value is 32 000.
• Description: This parameter allows setting the strictly high limit of concurrent Web processes of any type (contextual, non-contextual or belonging to the "pool of processes"—see selector 7, Maximum Web Process) supported by the 4D Web Server with 4D Developer 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.
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 Preferences dialog box (see the Web Services, Web Server Settings section).

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.

Note: If you pass a value inferior to the high limit of the "pool of processes," this limit is reduced in order to match the value of the selector 18. If necessary, the low limit of the pool (see selector 6, Minimum Web Process) is also modified.

Selector = 19 (Client Minimum process Web)
Selector = 20 (Client Maximum process Web)
Selector = 21 (Client Max Web requests size)
Selector = 22 (Client Port ID)
Selector = 23 (Client IP Address to listen)
Selector = 24 (Client Character set)
Selector = 25 (Client Max Concurrent Web Proc)
• Values: Identical to those of the corresponding 4D Developer or 4D Server selectors (see selectors 6 to 8, 15 to 18 and 27).
• Description: These selectors are used to specify the operating parameters of 4D Client machines used as Web servers.
The values defined using these selectors are applied to all the 4D Client machines used as Web servers. If you want to define values only for certain 4D Client machines, use the 4D Client Preferences dialog box.

Selector = 26 (Cache writing mode)
This selector has been disabled since it corresponds to a mechanism that has been optimized in 4D version 11.

Selector = 27 (Maximum Web requests size)
• Values: 500 000 to 2 147 483 648.
• 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.

Selector = 28 (4D Server Log Recording)
• Values: 0 or from 1 to X (0 = do not record, 1 to X = sequential number, added to the file name).

• 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).
4D Server lets you record each request received by the server machine in a log file. When this mechanism is enabled, the log file is created next to the database structure file. Its name is "4DRequestsLogX," where X is the sequential number of the log. Once the file reaches a size of 10 MB, it is closed and a new file 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.
This text file stores various information concerning each request in a simple tabulated format: time, process number, user, size of request, processing duration, etc. This information can be particularly useful when fine tuning the application or for statistical purposes. It can be imported, for example, into a spreadsheet software in order to be processed.

Note: It is possible to manually enable or disable the recording of requests using the **Ctrl+Alt+L** shortcut under Windows or the **Command+Option+L** shortcut under Mac OS.

Selector = *29* (Web Log Recording)
• 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.
• Description: Starts or stops the recording of Web requests received by the Web server of 4D Developer or 4D Server. By default, the value is 0 (requests not recorded).
The log of Web requests is stored as a text file named "logweb.txt" that is automatically placed next to the database 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.
This file can also be activated on the "Web/Advanced" page of the 4D Preferences.

WARNING: Formats 3 and 4 are custom formats whose contents must be set beforehand in the application Preferences on the "Web/Log Format" page. 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.

Selector = 30 (Client Web Log Recording)
• 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.
• 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).
The operation of this selector is identical to that of selector 29; however, it applies to all the 4D Client machines used as Web servers. If you only want to set values for certain client machines, use the Preferences dialog box of 4D Client.

Selector = 31 (Table Sequence Number)
• Values: Any longint value.
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).
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.

Selector = 32 (Real Display Precision)
• Values: Any positive longint value.
• Description: This selector lets you modify or get the number of non-significant digits truncated from the right by the real screen display algorithm. This value is set for the current application and session.
By default, the value of this option is 4. The value 0 indicates that the default value is used and that the parameter has not been modified during the session.
For historical reasons, 4D works with real numbers stored on 10 bytes and converts them to 8 bytes during processing (see the Display of Real Numbers section). This is entirely transparent and does not affect calculations; however certain results may not be displayed as anticipated. For example, the operation 4,1-4,09 displays the result 0.009999999999999780000, but searching for 0.01 finds the correct value.
Here is how 4D goes about displaying a real number: let's take the value 8.97499999999996158 obtained by a calculation as an example (the expected result would normally be 8.975). The algorithm which rounds off most accurately removes the last four digits (6158) by default and then checks whether the last remaining digit is a 0 or a 9. If it is 0, the algorithm goes back to the first 0 and removes all the others. If the value is 9, the algorithm goes back to the first 9 and rounds the decimal up to the next value. In our example, the value 8.974999999999996158 is thus transformed into 8.975.
It may happen that certain results end with 5 non-significant digits, like 8.974999999999986158 for instance. In this case, the algorithm cannot round the value off correctly because the last remaining digit (after removing the last four) is neither 0 nor 9 and it will thus do nothing.
You may want for the precision algorithm to truncate more or less digits according to the specific characteristics of your database. In this case, pass a custom value. Except for zero (4D internal value choice), this value will indicate the number of digits truncated by the precision algorithm.
Keep in mind that this setting does not affect the display of numbers, nor their internal processing.

• **Selector = 33** (TCP_NODELAY)
• Possible values: 0 or 1 (0 = disable, 1 = enable)
• Description: Enabling or disabling of the TCP_NODELAY network option. This option, which is internal to the TCP/IP protocol, controls a network communications optimization mechanism. It can be set separately for the server machine and the client machines. By default, the value is 1 (option enabled) on all the machines (server and clients).
In specific cases, in particular in the case of client/server connections by DSL or by VPN (Virtual Private Network), disabling this option can noticeably improve application performance. This operation should be carried out with caution and must be accompanied by load testing in the different client/server configurations.
If you modify this value, it will be necessary to restart the application in order for the new value to be taken into account.

• **Selector = 34** (Debug Log Recording)
• Possible values: 0, 1 or 2 (0 = do not record, 1 = record, 2 = record in detailed mode)
• Description: Starts or stops the sequential recording of events occurring at the 4D programming level, intended for debugging the application. By default, the value is 0 (events are not saved).
When this mode is enabled, various types of information can be recorded, more particularly:
- For each event, the number of milliseconds since the creation of the file and the process number (*[n]*),
- The execution of each 4D command (*cmd*) and each calling of a plug-in (*plugInName*); in this case, the stack level is indicated (*n*),
- Each calling of project methods (*meth*), object methods (*obj*) and form methods (*form*),
- When the detailed mode is activated (value = 2), additional information concerning the plug-ins are recorded: events in the plug-in areas (*EventCode*) and calls to 4D by the plug-ins (*externCall*).
The events are stored in a file named "4DDebugLog.txt" that is automatically placed next to the database structure file. Each event is systematically recorded in the file before its execution, which guarantees its presence in the file even when the application quits unexpectedly. Be careful, this file is erased and rewritten each time the application is launched.
This option can be activated for any type of 4D application (4D Developer, 4D Server, 4D Client, 4D Desktop), in interpreted or compiled mode.

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.

• **Selector = 35** (Client Server Port ID)
• Possible values: 0 to 65535

• Description: This parameter can be used to change the TCP port number where the 4D Server publishes the database (bound for 4D Client machines) by programming. By default, the value is 19813.
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.
The value is stored in the database structure file. It can be set with 4D Developer (single-user) but is only taken into account in client-server configuration.
When you modify this value, it is necessary to restart the server machine in order for the new value to be taken into account.

• **Selector = 36** (WEDD Signature)
• Possible values: String of 1 to 255 c haracters.
• Description: This parameter can be used to modify the WEDD signature of the open database (structure file and data file). By default, this signature is blank (the parameter is not defined). Keep in mind that the signature is case senstive.
The WEDD signature is used to associate a structure file with a specific data file. A structure file containing a WEDD signature can only be opened with the data file containing the same WEDD signature and vice versa. For more information about the WEDD signature, please refer to the *Design Reference* manual.
Setting this value by programming facilitates the distribution of upgrades for applications that have a custom signature.

Note: When you use this selector with the Get database parameter command, the string defined as the WEDD signature is returned in the optional stringValue parameter and the command returns 0.

• **Selector = 37** (Invert Objects)
• Possible values: 0, 1 or 2 (0 = mode disabled, 1 = automatic mode, 2 = mode enabled)
• 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 Database/Script Manager page of the application Preferences.
- Value 0 indicates that the mode is never enabled, regardless of the system configuration (corresponds to the No value in the Preferences).
- Value 1 indicates that the mode is enabled or disabled depending on the system configuration (corresponds to the Automatic value in the Preferences).
- Value 2 indicates that the mode is enabled, regardless of the system configuration (corresponds to the Yes value in the Preferences).
For more information, refer to the *Design Reference* manual of 4D.

• **Selector = 39** (HTTPS Port ID)
• Possible values: 0 to 65535
• Description: This selector can be used to modify by programming the TCP port number used by the 4D Developer and 4D Server Web server for secure connections via SSL (HTTPS protocol). The HTTPS port number is set on the "Web/Configuration" page of the Preferences dialog box. For more information, please refer to the Web Server Settings section.
By default, the value is 443 (standard value). You can use the constants of the "TCP Port Numbers" theme for the value parameter.

• **Selector = 40** (Client HTTPS Port ID)
• Possible values: 0 to 65535
• Description: 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).
This selector operates exactly the same way as selector 39; however, it applies to all the 4D Client machines used as Web servers. If you only want to modify the value of certain specific client machines, use the Preferences dialog box of 4D Client.

Selector = 41 (Unicode mode)
• Possible values: 0 (compatibility mode) or 1 (Unicode mode)
• 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.

Selector = 42 (Temporary memory size)
• Possible values: Positive longint > 50,000,000
• Description: Temporary memory size expressed in bytes. By default, this size is 50,000,000 (50 MB).
4D uses a special temporary memory that is devoted to indexing and sorting operations. The purpose of this memory is to protect the "standard" cache memory during mass operations. When the maximum size of temporary memory is reached (critical case), 4D automatically interrupts the last operation requested in order to avoid penalizing other processing underway. In most cases, the default configuration of the temporary memory will be perfectly adequate. However, in certain specific applications carrying out sorts or indexing in an especially intensive manner, increasing this memory size might help to noticeably improve performance. The ideal value will need to be determined empirically according to the specific features of the application and will generally be the result of real volumetric testing.

Selector = 43 (SQL Autocommit)
• Possible values: 0 (deactivation) or 1 (activation)
• Description: Activation or deactivation of the SQL auto-commit mode. By default, the value is 0 (deactivated mode)
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.

Selector = 44 (SQL Engine Case Sensitivity)
• Possible values: 0 (case not taken into account) or 1 (case-sensitive)
• Description: Activation or deactivation of case-sensitivity for string comparisons carried out by the SQL engine.
By default, the value is 1 (case-sensitive): the SQL engine differentiates between upper and lower case when comparing strings (sorts and queries). For example "ABC"= "ABC" but "ABC" # "Abc." 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").
This option can also be set on the SQL/Configuration page of the application Preferences.

Examples

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)
...
```

2. The IP address 192.193.194.195 will be set with the following statement:

```
SET DATABASE PARAMETER(IP Address to listen;0xC0C1C2C3)
```

3. This code can be used to switch the current character set and restart the database:

```
Current_unicode_mode:=Get database parameter(Unicode mode)
SET DATABASE PARAMETER(Unicode mode;1-Current_unicode_mode)
OPEN DATA FILE(Data file)
```

See Also

Get database parameter, QUERY SELECTION.

Structure file {(*)} → String

Parameter	Type		Description
*	*	→	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:MyCDs*f*: on the disk Macintosh HD, the command returns Macintosh HD:Documents:MyCDs*f*:MyCDs.

Note: In the particular case of a database that has been compiled and merged with 4D Desktop, this command returns the pathname of the application file (executable application) under Windows and Mac OS. Under Mac OS, 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 while running 4D Client, 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.

Examples

1. This example displays the name and the location of the structure file currently in use:

```
If (Application type#4D Client)
    $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.

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
```

See Also

Application file, COMPONENT LIST, Data file, DATA SEGMENT LIST.

VERIFY CURRENT DATA FILE{ (objects; options; method{; tablesArray; fieldsArray})}

Parameter	Type		Description
objects	Number	→	Objects to check
options	Number	→	Checking options
method	Text	→	Name of 4D callback method
tablesArray	Number array	→	Numbers of tables to be checked
fieldsArray	2D Number 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 Records+Verify Indexes (= value 0)
- options = 0 (log file is created)
- 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.

Note: This command must not be used when a process is creating or updating an index since, in this case, the verification results will not be valid.

See Also

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	Number	→	Objects to be checked
options	Number	→	Checking options
method	Text	→	Name of 4D callback method
tablesArray	Number array	→	Numbers of tables to be checked
fieldsArray	2D Number array	→	Numbers of indexes to be checked

Description

TheVERIFY 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 check whether the data file is open, use the VERIFY CURRENT DATA FILE command. If you attempt to check 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:
- Verify Records (4)
- Verify Indexes (8)
- Verify All No Callback (16)
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 No Callback option is a special case. It carries out complete internal verification but, for internal reasons, does not allow callback methods. This verification is nevertheless compatible with the creation of a log.

• The options parameter is used to set verification options. A single option is currently available, found in the"Data file maintenance" theme: Do not create log file (16384). 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 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 OK	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

Object type: When an object has been verified, an OK 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, 3or 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.

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). Its name is based on that of the data file and it is placed next to this file. For example, for a data file named "data.4dd," the log file will be named "data_verify_log.xml."

Examples

1. Simple checking of data and indexes:

```
VERIFY DATA FILE($StructName;$DataName;Verify Indexes+Verify Records;
                                                Do not create log file;"")
```

2. Complete verification with log file:

```
VERIFY DATA FILE($StructName;$DataName;Verify All No Callback;0;"")
```

3. Checking of records only:

```
VERIFY DATA FILE($StructName;$DataName;Verify Records;0;"")
```

4. Checking of records from tables 3 and 7 only:

```
ARRAY LONGINT($arrTableNums;2)
ARRAY LONGINT($arrIndex;0) `not used but mandatory
$arrTableNums{1}:=3
$arrTableNums{2}:=7
VERIFY DATA FILE($StructName;$DataName;Verify Records;0;"FollowScan";
                                                    $arrTableNums;$arrIndex)
```

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)
```

See Also

VERIFY CURRENT DATA FILE.

System Variables or Sets

If the callback method does not exist, an error is generated and the system variable OK is set to 0.

Version type

Version type → Long Integer

Parameter	Type		Description
This command does not require any parameters			
Function result	Long Integer	←	0 -> Full version 1 -> Demo Limited version

Description

The Version type command returns a numeric value that denotes the type of 4D environment version that you are running. 4D provides the following predefined constants:

Constant	Type	Value
Full Version	Long Integer	0
Demo Version	Long Integer	1

Example

Your 4D application includes some features that are not available when a demo version of the 4D environment is used. Surround these features with a test that calls Version type:

```
If (Version type=Full Version)
   ` Perform appropriate operations
Else
   ALERT("This feature is not available in the Demo version of"+
                                          " Super Management Systems™.")
End if
```

See Also

Application type, Application version.

4

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 section Creating Arrays.

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 section Using the element zero of an array.

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 sections Arrays and the 4D Language and Arrays and Pointers.

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 sections Arrays and Form Objects and Grouped Scrollable Areas.

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 section Arrays and Memory.

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 (from 0 to 32,000 characters per element) (**)
ARRAY STRING	String values (from 0 to 255 characters per element) (**)
ARRAY DATE	Date values
ARRAY BOOLEAN	Boolean values
ARRAY PICTURE	Pictures values
ARRAY POINTER	Pointer values

Each array declaration command can create or resize one-dimensional or two-dimensional arrays. For more information about two-dimensional arrays, see the section Two-dimensional Arrays.

(*) Longint arrays allows you to manipulate data of Time type. To display a Time array in a form, apply to the associated form object the display format &/x, in which x represents the number of the format in the Time formats list (by order of appearance). For example, &/4 will display the Hour Min format.

(**) The difference between Text arrays and String arrays lies in the nature of their elements. In both types of array, elements can hold text values (characters). However:

- In a Text array, each element is of variable length and stores its characters in a separate part of memory.
- In a String array, all elements have the same fixed length (the length passed when the array was created). All elements are stored one after the other in the same part of memory, no matter what the contents.

Due to this structural difference, string arrays act faster than text arrays. Note, however, that an element of a String array can only hold up to 255 characters.

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 INSERT LISTBOX ROW.

• You can create graphs and charts on series of values stored in tables, subtables, and arrays. For more information, see the GRAPH command.

• Although version 6 brings a full set of new commands to work with hierarchical lists, the commands LIST TO ARRAY and ARRAY TO LIST (from the previous version) have been retained for compatibility.

• 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.

See Also

ARRAY BOOLEAN, ARRAY DATE, ARRAY INTEGER, ARRAY LONGINT, ARRAY PICTURE, ARRAY POINTER, ARRAY REAL, ARRAY STRING, ARRAY TEXT, Arrays, Two-dimensional Arrays.

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:

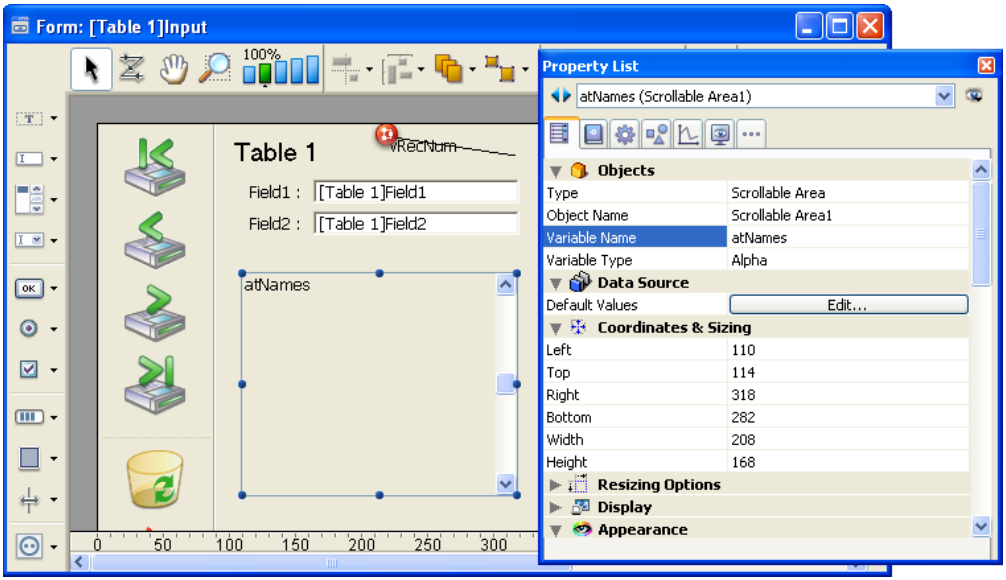
- Pop-up/Drop-down List
- Combo Box
- Scrollable Area
- Tab Control
- List box

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.

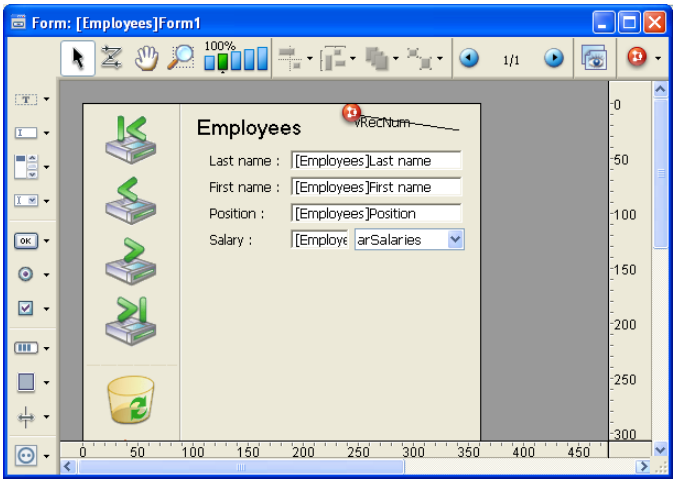
Notes:
• You cannot display two-dimensional arays or pointer arrays.
• The management of **List box** type objects (which may contain several arrays) entails many specific aspects. These particularities are covered in the Managing List Box Objects section.

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.

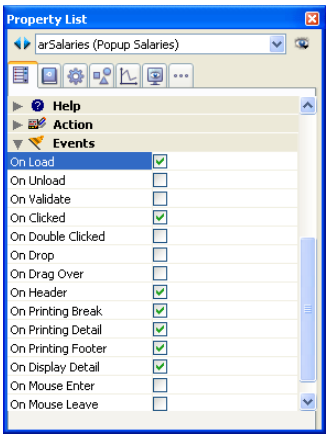
Create the arSalaries drop-down list on a form

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.

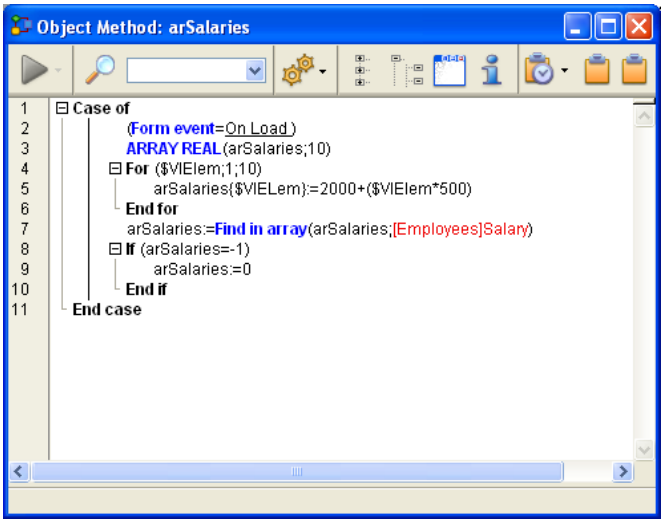


Initializing the array

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.

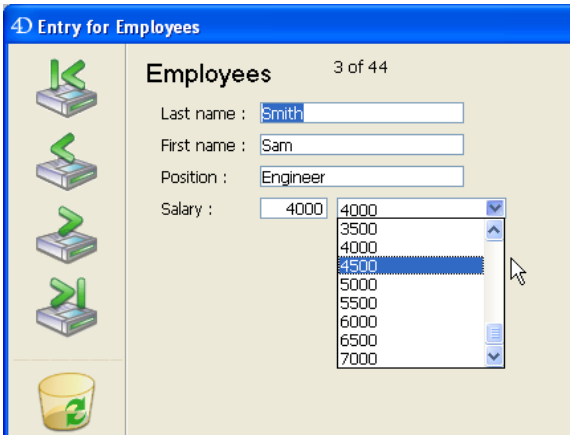
Note: For more information about the **array selected element**, read the next section.

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:

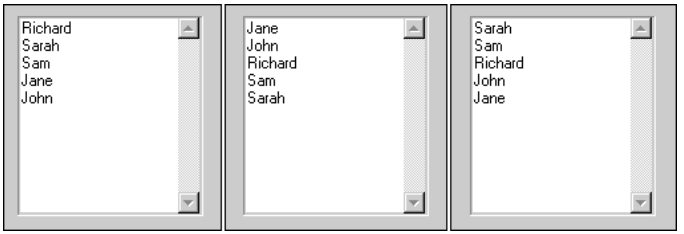
```
SORT ARRAY(atNames;>)
```

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 New list 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

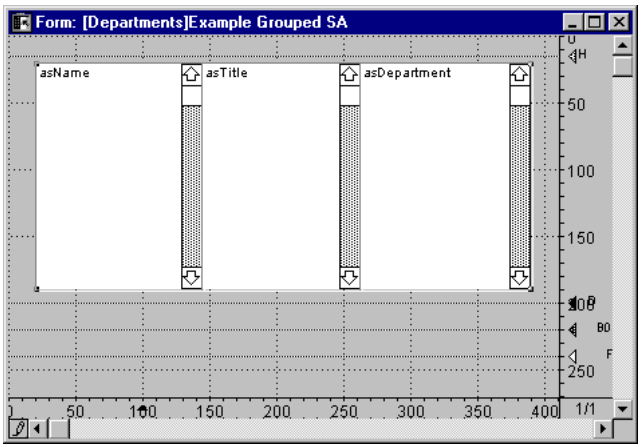
```
      ` asColors Combo Box object method
Case of
   : (Form event=On Load)
      ARRAY STRING(31;asColors;3)
      asColors{1}:="Blue"
      asColors{2}:="White"
      asColors{3}:="Red"
   : (Form event=On Clicked)
      If (asColors{0}#"")
            ` The object automatically changes its value
            ` 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
```

See Also

Arrays, Grouped Scrollable Areas.

Compatibility note: Grouped scrollable areas can still be used in 4D; however, starting with version 2004 they can be replaced by List box type objects. For more information about this, refer to the Overview of List boxes section.

You can group scrollable areas for display in a form. When several scrollable areas are grouped, they act as one scrollable area. Each scrollable area can have its own font and style; however, we recommend that you use the same font height (which depends on the font and font size) for each column. When displayed during data entry, only the frontmost scrollable area displays a scroll bar. Following are three scrollable areas grouped together in the Design environment:

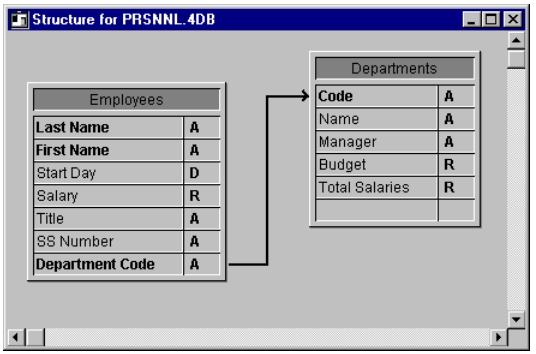


Here are some tips on creating grouped scrollable areas:
• Make sure that all the arrays have been given the same size (number of elements).
• Use the same font size for each area.
• Make each area the same height.
• Align the tops of all the areas.
• Make sure the areas do not overlap.
• Make sure that the area on the right is in front, because the scroll bar appears on the frontmost area.
• Group the areas (using the Group menu command) to make them work as one scrollable area.

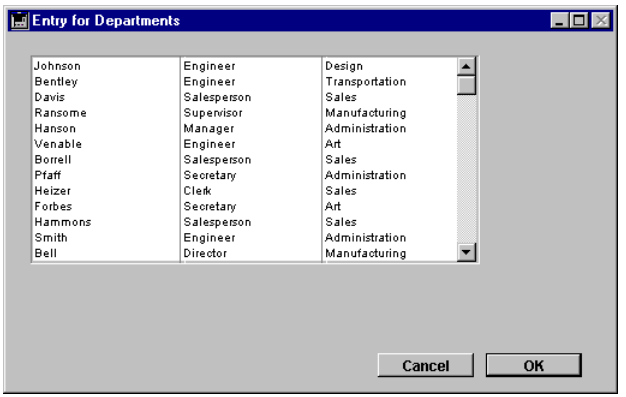
The following project method fills the three arrays and displays them on the screen:

ALL RECORDS(Employees)
SELECTION TO ARRAY([Employees]Last Name;asName;[Employees]Title;asTitle;[Departments]Name;asDepartment)
DIALOG([Departments];"Example Grouped SA")

This method uses the data in the fields of the [People] table and the [Departments] table. These tables are shown here:



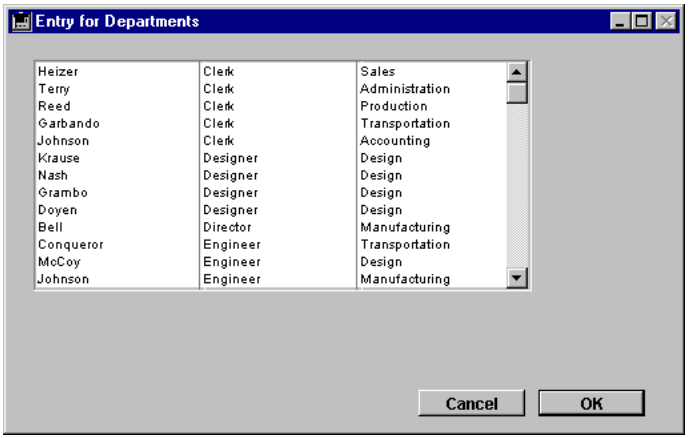
Note: The [Departments] table can be used, provided that there is an automatic relation from [People] to [Departments].
The resulting display:



Note that only a single scroll bar is displayed; it is always on the frontmost scrollable area. This scroll bar controls the scrolling of all three arrays as if they were one. When the user clicks a line, all three areas are highlighted simultaneously. The variable associated with each scrollable area is set to the number of the line that the user clicks; only the object method for the area that is clicked executes. For example, if the user clicks the name “Bentley,” asName, asTitle, and asDepartment are all set to two, but only the object method for asName executes. If you set the selected element of one of the arrays in the grouped scrollable areas, the other arrays are set to the same selected element for the next event, and the respective line in the scrollable area is highlighted.

The arrays can be sorted with the command SORT ARRAY. For example:

SORT ARRAY(asTitle;asName;asDepartment;>)
The following is the result of the sort:



Note that the arrays were sorted based on the first argument to the SORT ARRAY command; the other two arrays were specified in order to keep the rows synchronized. The command SORT ARRAY always sorts the arrays (if several are specified) on the values of the first array and keeps the additional arrays synchronized.

Note: SORT ARRAY does not perform a multi-level sort on arrays. To show a table similar to the one above and also perform multi-level sorts (i.e., by department, then by title, then by name), use a subform in which you display the table, and then use ORDER BY.

See Also

Arrays, Arrays and Form Objects.

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 section Arrays and Pointers.

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.

See Also

Arrays, 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.

Note: You can pass process and interprocess arrays as parameters, but not local arrays.

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
```

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.

See Also

Arrays, Arrays and the 4D Language.

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 section Arrays and Form Objects.

Here are two other examples.

1. If you want to execute an action only when you click on an element other than the previously selected element, 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
```

2. In ASCII compatibility mode, when sending or receiving a stream of characters to or from a document or a serial port, 4D provides a way to filter ASCII codes between platforms and systems that use different ASCII maps— the commands USE CHARACTER SET, Mac to ISO, ISO to Mac, Mac to Win and Win to Mac.

In certain cases, you might want to fully control the way ASCII codes are translated. One way to do this is to use an Integer array of 255 elements, where the Nth element is set to the translated ASCII code for the character whose source ASCII code is N. For example, if the ASCII code #187 must be translated as #156, you would write <>aiCustomOutMap{187}:=156 and <>aiCustomInMap{156}:=187 in the method that initializes the interprocess arrays used everywhere in the database. You can then send a stream of characters with the following custom project method:

```
    ` X SEND PACKET ( Text { ; Time } )
For ($vlChar;1;Length($1))
    $1[[vlChar]]:=Char(<>aiCustomOutMap{Character code($1[[vlChar]])})
End for
If (Count parameters>=2)
    SEND PACKET ($2;$1)
Else
    SEND PACKET ($1)
End if

    ` X Receive packet ( Text { ; Time } ) -> Text
If (Count parameters>=2)
    RECEIVE PACKET ($2;$1)
Else
    RECEIVE PACKET ($1)
End if
$0:=$1
For ($vlChar;1;Length($1))
    $0[[vlChar]]:=Char(<>aiCustomInMap{Character code($0[[vlChar]])})
End for
```

In this advanced example, if a stream of characters containing NULL characters (ASCII code zero) is sent or received, the zero element of the arrays <>aiCustomOutMap and <>aiCustomInMap will play its role as any other element of the 255 element arrays.

See Also

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.

See Also

Arrays.

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) * Declared length (+1 of odd, +2 if even)
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) * 6 + Sum of the size of each text
Two-dimemsional	(1+number of elements) * 12 + Sum of the size of each array

Note: 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
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
```

See Also

Arrays, ON ERR CALL.

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)
```

See Also

DELETE FROM ARRAY, INSERT IN ARRAY.

ARRAY BOOLEAN (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	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.

Examples

1. This example creates a process array of 100 Boolean elements:

```
ARRAY BOOLEAN (abValues;100)
```

2. This example creates a local array of 100 rows of 50 Boolean elements:

```
ARRAY BOOLEAN ($abValues;100;50)
```

3. This example creates an interprocess array of 50 Boolean elements and sets each even element to True:

```
ARRAY BOOLEAN (<>abValues;100)
For ($vlElem;1;50)
    <>abValues{$vlElem}:=(($vlElem%2)=0)
End for
```

See Also

ARRAY INTEGER.

ARRAY DATE (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	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 ARRAY DATE 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.

Examples

1. This example creates a process array of 100 Date elements:

```
ARRAY DATE (adValues;100)
```

2. This example creates a local array of 100 rows of 50 Date elements:

```
ARRAY DATE ($adValues;100;50)
```

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 (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	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.

Examples

1. This example creates a process array of 100 2-byte Integer elements:

```
ARRAY INTEGER (aiValues;100)
```

2. This example creates a local array of 100 rows of 50 2-byte Integer elements:

```
ARRAY INTEGER ($aiValues;100;50)
```

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
```

See Also

ARRAY LONGINT, ARRAY REAL.

ARRAY LONGINT (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The ARRAY LONGINT command creates and/or resizes an array of 4-byte Long 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.

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.

Examples

1. This example creates a process array of 100 4-byte Long Integer elements:

```
ARRAY LONGINT (alValues;100)
```

2. This example creates a local array of 100 rows of 50 4-byte Long Integer elements:

```
ARRAY LONGINT ($alValues;100;50)
```

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
```

See Also

ARRAY INTEGER, ARRAY REAL.

ARRAY PICTURE (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array, or Number of rows if size2 is specified
size2	Number	→	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 ARRAY PICTURE 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.

Examples

1. This example creates a process array of 100 Picture elements:

```
ARRAY PICTURE (agValues;100)
```

2. This example creates a local array of 100 rows of 50 Picture elements:

```
ARRAY PICTURE ($agValues;100;50)
```

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="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 (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array, or Number of rows if size2 is specified
size2	Number	→	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.

Examples

1. This example creates a process array of 100 Pointer elements:

```
ARRAY POINTER (apValues;100)
```

2. This example creates a local array of 100 rows of 50 Pointer elements:

```
ARRAY POINTER ($apValues;100;50)
```

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 for
```

ARRAY REAL (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	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.

Examples

1. This example creates a process array of 100 Real elements:

```
ARRAY REAL (arValues;100)
```

2. This example creates a local array of 100 rows of 50 Real elements:

```
ARRAY REAL ($arValues;100;50)
```

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
```

See Also

ARRAY INTEGER, ARRAY LONGINT.

ARRAY STRING (strLen; arrayName; size{; size2})

Parameter	Type		Description
strLen	Number	→	Length of string (1... 255)
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The ARRAY STRING command creates and/or resizes an array of String elements in memory.

• The strLen parameter specifies the maximum number of characters that can be contained in each array element in a string array. The length can be from 1 to 255 characters.
• 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 STRING 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.

Examples

1. This example creates a process array of 100 31-character String elements:

```
ARRAY STRING (31;asValues;100)
```

2. This example creates a local array of 100 rows of 50 63-character String elements:

```
ARRAY STRING (63;$asValues;100;50)
```

3. This example creates an interprocess array of 50 255-character String elements and sets each element to the value “Element #” followed by its element number:

```
ARRAY STRING (255;<>asValues;50)
For ($vlElem;1;50)
    <>asValues{$vlElem}:="Element #"+String($vlElem)
End for
```

See Also

ARRAY TEXT.

ARRAY TEXT (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	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.

Examples

1. This example creates a process array of 100 Text elements:

```
ARRAY TEXT (atValues;100)
```

2. This example creates a local array of 100 rows of 50 Text elements:

```
ARRAY TEXT ($atValues;100;50)
```

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
```

See Also

ARRAY STRING.

ARRAY TO LIST

version 3

Compatibility Note
Due to the new implementation of Choice Lists, compatibility for this command could not be fully maintained. Also, starting with version 6, we recommend that you use the command SAVE LIST to work with the hierarchical lists defined in the Design environment List Editor.

ARRAY TO LIST (array; list{; itemRefs})

Parameter	Type		Description
array	Array	→	Array from which to copy array elements
list	String	→	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 list list (as defined in the Design environment List Editor) using the elements of the array array.

This command allows you to define only the first level items of the list.

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: In the previous version of 4D, this parameter was used to link other lists to each element in array. If an element of the links array was the name of an existing list, then that list was attached to the corresponding item.

You can continue to use ARRAY TO LIST to build a list based on the elements of an array. However, this command does not provide a means of working with the child items. To work with hierarchical lists, use the new Hierarchical Lists commands introduced in version 6.

Example

The following example copies the array atRegions to the list called “Regions:”

```
ARRAY TO LIST (atRegions;"Regions")
```

See Also

LIST TO ARRAY, Load list, ON ERR CALL, SAVE LIST.

Error Handling

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.

version 11 (Modified)

ARRAY TO SELECTION (array; field{; array2; field2; ...; arrayN; fieldN})

Parameter	Type		Description
array	Array	→	Array to copy to the selection
aField	Field	←	Field to receive the array data

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.

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.

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.

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

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)
```

See Also

SELECTION TO ARRAY.

BOOLEAN ARRAY FROM SET

version 6.5

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.

See Also

CREATE SET FROM 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.

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) → Longint

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")
```

See Also

Find in array.

DELETE FROM ARRAY (array; where{; howMany})

Parameter	Type		Description
array	Array	→	Array from which to delete elements
where	Number	→	Element at which to begin deletion
howMany	Number	→	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.

Examples

1. The following example deletes three elements, starting at element 5:

```
DELETE FROM ARRAY(anArray; 5; 3)
```

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
```

See Also

INSERT IN ARRAY, Size of array.

DISTINCT VALUES (aField; array)

Parameter	Type		Description
aField	Field	→	Indexable field to use for data
array	Array	←	Array to receive field data

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.

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.

If you create the array prior to the call, DISTINCT VALUES expects an array type compatible with the field you pass. Otherwise, in interpreted mode, DISTINCT VALUES will create an array of the proper type. However, if the field or subfield is of type Time, the command expects or creates a LongInt array.

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.

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.

Examples

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.")
```

2. The following example returns in asKeywords all the keywords that are attached (using a subtable) to the 4D Write documents stored in the table [Documentation] and whose theme is “Economy”:

```
QUERY ([Documentation];[Documentation]Theme="Economy")
DISTINCT VALUES([Documentation]Keywords'Keyword;asKeywords)
```

After this array has been built, you can reuse it to quickly locate all the documents associated with the selected keyword:

```
QUERY ([Documentation];[Documentation]Keywords'Keyword=asKeywords{asKeywords})
SELECTION TO ARRAY ([Documentation]Subject;asSubjects)
   ` ...
```

See Also

ON ERR CALL, SELECTION RANGE TO ARRAY, SELECTION TO ARRAY.

Find in array

version 3

Find in array (array; value{; start}) → Number

Parameter	Type		Description
array	Array	→	Array to search
value	Expression	→	Value of same type to search in the array
start	Number	→	Element at which to start searching
Function result	Number	←	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.

Examples

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)
```

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
```

See Also

DELETE FROM ARRAY, INSERT IN ARRAY, Size of array.

INSERT IN ARRAY (array; where{; howMany})

Parameter	Type		Description
array	Array	→	Name of the array
where	Number	→	Where to insert the elements
howMany	Number	→	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.

Examples

1. The following example inserts five new elements, starting at element 10:

```
INSERT IN ARRAY(anArray;10;5)
```

2. The following example appends an element to an array:

```
$vlElem:=Size of array(anArray)+1
INSERT IN ARRAY (anArray;$vlElem)
anArray{$vlElem}:=...
```

See Also

DELETE FROM ARRAY, Size of array.

version 3

Compatibility Note
Due to the new implementation of Choice Lists, compatibility for this command could not be fully maintained. Also, starting with version 6, we recommend that you start using the command Load list to work with the hierarchical lists defined in the Design environment List Editor.

LIST TO ARRAY (list; array{; itemRefs})

Parameter	Type		Description
list	String	→	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 list.

LIST TO ARRAY creates or overrides an array with a new text array.

The optional itemRefs parameter (a numeric array) returns the list item reference numbers.

Compatibility Note: In the previous version of 4D, this array was filled with the names of any linked lists. If an element of the list had a linked list, the name of the linked list was put into the array element with the same number as the list element. If there was no linked list, then the element was the empty string. The second array was set to the same size as array. You could use the names in this array to access the linked lists.

You can continue to use LIST TO ARRAY to build an array based on the first level items of a hierarchical list. However, this command does not provide you with the child items, if any. To work with hierarchical lists, use the new Hierarchical Lists commands introduced in version 6.

Example

The following example copies the items of a list called Regions into an array called atRegions:

```
LIST TO ARRAY ("Regions"; atRegions )
```

See Also

ARRAY TO LIST, Load list, SAVE LIST.

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.

See Also

CREATE SELECTION FROM ARRAY.

MULTI SORT ARRAY (array{; sort}{; array2; sort2; ...; arrayN; sortN})

Parameter	Type		Description
array	Array	→	Array(s) to be sorted
sort	> or <	→	> 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 function is particularly useful in the context of grouped scrolling areas in forms.

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.

Examples

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;>)
```

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;>).

See Also

ORDER BY, SELECTION TO ARRAY, SORT ARRAY.

SELECTION RANGE TO ARRAY

version 3.5.3

SELECTION RANGE TO ARRAY (start; end; field | table; array{; field2 | table2; array2; ...; fieldN | tableN; arrayN})

Parameter	Type		Description
start	Number	→	Selected record number where data retrieval starts
end	Number	→	Selected record number where data retrieval ends
aField | aTable	Field or 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. There are two exceptions:

- If a Text field is copied into a String array. In this case, the array will remain a String array.
- A Time field is copied into a Long Integer array.

Note: You cannot specify Subtable fields or subfields.

If you load record numbers, they are copied into a Long Integer array.

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.

Examples

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)
```

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)
```

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
```

See Also

ON ERR CALL, SELECTION TO ARRAY, SET AUTOMATIC RELATIONS.

SELECTION TO ARRAY

version 3

SELECTION TO ARRAY (field | table; array{; field2 | table2; array2; ...; fieldN | tableN; arrayN})

Parameter	Type		Description
aField | aTable	Field or 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

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 specified in the first parameter. SELECTION 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, 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. There are two exceptions:

- If a Text field is copied into a String array, the array will remain a String array.
- A Time field is copied into a Long Integer array.

Note: You cannot specify Subtable fields or subfields.

If you load record numbers, they are copied into a Long Integer array.

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 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.

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.

Examples

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)
```

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)
```

See Also

ARRAY TO SELECTION, MULTI SORT ARRAY, ON ERR CALL, SELECTION RANGE TO ARRAY, SET AUTOMATIC RELATIONS.

Size of array

version 3

Size of array (array) → Number

Parameter	Type		Description
array	Array	→	Array whose size is returned
Function result	Number	←	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
```

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
```

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}
```

See Also

DELETE FROM ARRAY, INSERT IN ARRAY.

SORT ARRAY (array{; array2; ...; arrayN}{; > or <})

Parameter	Type		Description
array	Array	→	Arrays to sort
> or <		→	> 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. This feature is especially useful with grouped scrollable areas in a form; SORT ARRAY maintains the synchronicity of the arrays that sustain the scrollable areas.

Examples

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)
```

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;<)
```

See Also

MULTI SORT ARRAY, ORDER BY, SELECTION TO ARRAY.

5

Backup

BACKUP

Parameter	Type	Description
This command 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 application Preferences. 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.

See Also

GET BACKUP INFORMATION, On Backup Startup Database Method, RESTORE.

System Variables or Sets

If the backup is performed correctly, the system variable OK is set to 1; otherwise, it is set to 0.

Error Handling

In case of any incidents, an error is generated which you can intercept by means of an error-handling method installed using the ON ERR CALL command.

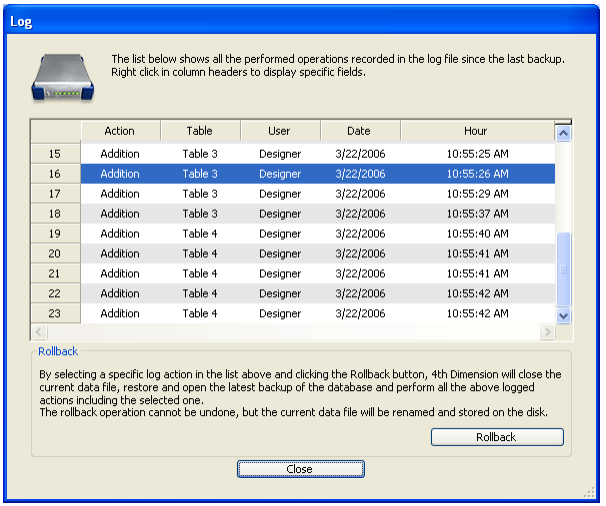
CHECK LOG FILE

Parameter	Type	Description

This command 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 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.

Note: The **Check Log File...** menu command is always available on the server machine.

See Also

Backup management system errors, RESTORE.

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.
• If this command is called when a transaction or indexing operation is underway in one of the active processes, its execution will be delayed for a period set on the Backup/Backup page of the application Preferences. If, at the end of this waiting period, the operation is still not terminated, the execution of this command is cancelled and the error 1422 is returned.
You can intercept these errors using an error-handling method installed with the ON ERR CALL command.

GET BACKUP INFORMATION (selector; info1; info2)

Parameter	Type		Description
selector	Longint	→	Type of information to get
info1	Date | Integer	←	Value 1 of the selector
info2	Time | String	←	Value 2 of the selector

Description

The GET BACKUP INFORMATION command allows getting 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.

See Also

RESTORE.

GET RESTORE INFORMATION

version 2004

GET RESTORE INFORMATION (selector; info1; info2)

Parameter	Type		Description
selector	Longint	→	Type of information to get
info1	Date | Integer	←	Value 1 of the selector
info2	Time | String	←	Value 2 of the selector

Description

The GET RESTORE INFORMATION command allows getting 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.

See Also

RESTORE.

SELECT LOG FILE (logFile | *)

Parameter	Type		Description
logFile | *	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 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.

See Also

ON ERR CALL.

System Variables and Sets

OK is set to 1 if the log file is correctly created, or closed.

Error Handling

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.

INTEGRATE LOG FILE (pathName)

Parameter	Type		Description
pathName	Text	→	Name or pathname of the log file to be integrated

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.

Description

The INTEGRATE LOG FILE command integrates the log file, whose name or pathname was passed in the pathName parameter, into the current database. Afterwards, the file that was integrated becomes the new current log file of the database. 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).

Only log files that are not filed (extension .4DL) can be integrated using this command. No dialog box appears; but a progress bar is displayed on screen.

In the pathName parameter, you can pass an absolute pathname or one that is relative to the database folder. If you pass an empty string in this parameter, a standard open file dialog box will be displayed to allow you to indicate the file to be integrated. If this dialog box is cancelled, no file will be integrated and the system variable OK is set to 0.

When using this command, it is up to the developer to:

• Install the mirror database on the mirror machine and make sure that the data file will not be modified other than by the integration of the log file using the INTEGRATE LOG FILE command. In order to detect whether it is the mirror version of the database, it is possible to place a file in the 4D Extensions folder or database folder and to test for its presence for instance during the On Startup database method. If the file is present, the mirror mode is activated.

• Set up a communication system between the operational database and the mirror database in order to organize the sending and receiving of the log file segments. To do this, it is possible to use a Web service, the 4D Open for 4D plug-in or the 4D Internet Commands.

• Handle any possible transmission errors that may occur between the two databases.

See Also

New log file.

System Variables or Sets

If the integration is carried out correctly, the system variable OK is set to 1; otherwise, it is set to 0.

Error Handling

In the event of an error, the command generates a code that can be intercepted using the ON ERR CALL command. If there are any locked records in the database, the command does nothing and the error 1420 is generated.

Log File → String

Parameter	Type		Description
This command does not require any parameters			
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

See Also

SELECT LOG FILE.

System Variables or Sets

If the database is operating without a log file, the system variable OK is set to 0; otherwise, it is set to 1.

New log file

New log file → Text

Parameter	Type		Description
This command does not require any parameters			
Function result	Text	←	Full pathname of closed log file

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.

Description

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].4DL. 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[0004-0000].4DL.
• 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[0003-0005].4DL.

Before performing this operation, 4D Server checks that no other critical operation (transaction or indexing) is underway. If a critical operation is underway, 4D Server respects the waiting times set on the Backup page in the Backup theme of the Preferences.

See Also

INTEGRATE LOG FILE.

Error Handling

In the event of an error, the command generates a code that can be intercepted using the ON ERR CALL command.

On Backup Shutdown Database Method

version 11 (Modified)

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 Developer, 4D Server, 4D Client, 4D Desktop and 4D Interpreted Desktop as well as 4D applications compiled and merged with 4D Unlimited 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.
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)

See Also

BACKUP, On Backup Startup Database Method.

On Backup Startup Database Method

version 2004

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 Developer, 4D Server, 4D Client, 4D Desktop and databases merged with 4D 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).

See Also

BACKUP, On Backup Shutdown Database Method.

RESTORE

Parameter	Type	Description

This command does not require any parameters

Description

The RESTORE command displays a standard Open file dialog box that can be used by the user to select an archive to restore.

This command is useful with customized interfaces for managing backups.

Note: In a 4D application that is compiled and merged with *4D Unlimited Desktop*, the RESTORE command causes the display of a standard open file dialog box that lists by default any files having the “4BK” extension.

See Also

BACKUP, GET RESTORE INFORMATION.

6

BLOB

Definition

4D supports the BLOB (Binary Large OBjects) data type.

You can define BLOB fields and BLOB variables:

• 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.

Note: There is no array for BLOBs.

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 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 (Picture and subtable field types), 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.
• 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 SET RESOURCE enable you to work with any type stored of resource stored on disk. Refer to the Resources chapter for more information.
• SEND HTML 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 Secured Protocol chapter for more information.

BLOB PROPERTIES

version 6.0

BLOB PROPERTIES (blob; compressed{; expandedSize{; currentSize}})

Parameter	Type		Description
blob	BLOB	→	BLOB for which to get information
compressed	Number	←	0 = BLOB is not compressed 1 = BLOB compressed compact 2 = BLOB compressed fast
expandedSize	Number	←	Size of BLOB (in bytes) when not compressed
currentSize	Number	←	Current size of BLOB (in bytes)

Description

The BLOB PROPERTIES command returns information about the BLOB blob.

• The compressed parameter tells whether or not the BLOB is compressed, and returns one of the following values. Note: 4D provides the predefined constants.

Constant	Type	Value
Is not compressed	Long Integer	0
Compact compression mode	Long Integer	1
Fast compression mode	Long Integer	2

• 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.

Examples

1. See examples for the commands COMPRESS BLOB and EXPAND BLOB.

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
` 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.")
```

See Also

COMPRESS BLOB, EXPAND BLOB.

BLOB size

version 6.0

BLOB size (blob) → Longint

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.

Examples

The line of code adds 100 bytes to the BLOB myBlob:

```
SET BLOB SIZE (BLOB size(myBlob)+100)
```

See Also

SET BLOB SIZE.

BLOB TO DOCUMENT (document; blob{; *})

Parameter	Type		Description
document	String	→	Name of the document
blob	BLOB	→	New contents for the document
*	*	→	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).

Note 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. 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 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
        ` Write the document contents
    BLOB TO DOCUMENT (Document;[YourTable]YourBLOBField)
    If (OK=0)
        ` Handle error
    End if
End if
```

See Also

Create document, DOCUMENT TO BLOB, Open document.

System Variables

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

version 6.0

BLOB to integer (blob; byteOrder{; offset}) → Number

Parameter	Type		Description
blob	BLOB	→	BLOB from which to get the integer value
byteOrder	Number	→	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	Number	←	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
Native byte ordering	Long Integer	0
Macintosh byte ordering	Long Integer	1
PC byte ordering	Long Integer	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
```

See Also

BLOB to longint, BLOB to real, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

BLOB to list

version 6.0

BLOB to list (blob{; offset}) → ListRef

Parameter	Type		Description
blob	BLOB	→	BLOB containing a hierarchical list
offset	Number	→	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
```

See Also

LIST TO BLOB.

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

version 6.0

BLOB to longint (blob; byteOrder{; offset}) → Number

Parameter	Type		Description
blob	BLOB	→	BLOB from which to get the Long Integer value
byteOrder	Number	→	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	Number	←	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
Native byte ordering	Long Integer	0
Macintosh byte ordering	Long Integer	1
PC byte ordering	Long Integer	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
```

See Also

BLOB to integer, BLOB to real, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

BLOB to real (blob; realFormat{; offset}) → Real

Parameter	Type		Description
blob	BLOB	→	BLOB from which to get the Real value
realFormat	Number	→	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
Native real format	Long Integer	0
Extended real format	Long Integer	1
Macintosh double real format	Long Integer	2
PC double real format	Long Integer	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
```

See Also

BLOB to integer, BLOB to longint, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

BLOB to text (blob; textFormat{; offset{; textLength}}) → Text

Parameter	Type		Description
blob	BLOB	→	BLOB from which to get the text
textFormat	Number	→	Format and character set of text
offset	Variable	→	Offset within the BLOB (expressed in bytes)
		←	New offset after reading
textLength	Number	→	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	Long Integer	0
Mac Pascal string	Long Integer	1
Mac Text with length	Long Integer	2
Mac Text without length	Long Integer	3
UTF8 C string	Long Integer	4
UTF8 Text with length	Long Integer	5
UTF8 Text without length	Long Integer	6

Notes:
- The "UTF8" constants can only be used when the application runs in Unicode mode.
- The Mac Text with length constant 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 andUTF8 Text without length, for which you MUST specify the number of characters to be read in the parameter textLength. 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.

See Also

BLOB to integer, BLOB to longint, BLOB to real, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

BLOB TO VARIABLE (blob; variable{; offset})

Parameter	Type		Description
blob	BLOB	→	BLOB containing 4D variables
variable	Variable	←	Variable to write with BLOB contents
offset	Number	→	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.

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 command VARIABLE TO BLOB.

See Also

VARIABLE TO BLOB.

System Variables or Sets

The OK variable is set to 1 if the variable has been successfully rewritten, otherwise it is set to 0.

COMPRESS BLOB (blob{; compression})

Parameter	Type		Description
blob	BLOB	→	BLOB to compress
compression	Number	→	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 the internal 4D 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:
• If you pass 1, the BLOB is compressed as much as possible, at the expanse of the speed of compression and decompression operations.
• If you pass 2, the BLOB is 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).
• If you pass another value or if you omit the parameter, the BLOB is compressed as much as possible, using the compression mode 1.

4D provides the following predefined constants:

Constant	Type	Value
Compact compression mode	Long Integer	1
Fast compression mode	Long Integer	2

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.

Examples

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.

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
```

See Also

BLOB PROPERTIES, EXPAND BLOB.

System Variables or Sets

The OK variable is set to 1 if the BLOB has been successfully compressed; otherwise, it is set to 0.

COPY BLOB

version 6.0

COPY BLOB (srcBLOB; dstBLOB; srcOffset; dstOffset; len)

Parameter	Type		Description
srcBLOB	BLOB	→	Source BLOB
dstBLOB	BLOB	→	Destination BLOB
srcOffset	Variable	→	Source position for the copy
dstOffset	Variable	→	Destination position for the copy
len	Number	→	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.

See Also

DELETE FROM BLOB, INSERT IN 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.

See Also

ENCRYPT BLOB, GENERATE ENCRYPTION KEYPAIR.

DELETE FROM BLOB

version 6.0

DELETE FROM BLOB (blob; offset; len)

Parameter	Type		Description
blob	BLOB	→	BLOB from which to delete bytes
offset	Number	→	Starting offset where bytes will be deleted
len	Number	→	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.

See Also

INSERT IN BLOB.

DOCUMENT TO BLOB

version 6.0

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
*	*	→	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
```

See Also

BLOB TO DOCUMENT, Open document.

System Variables

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 (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).

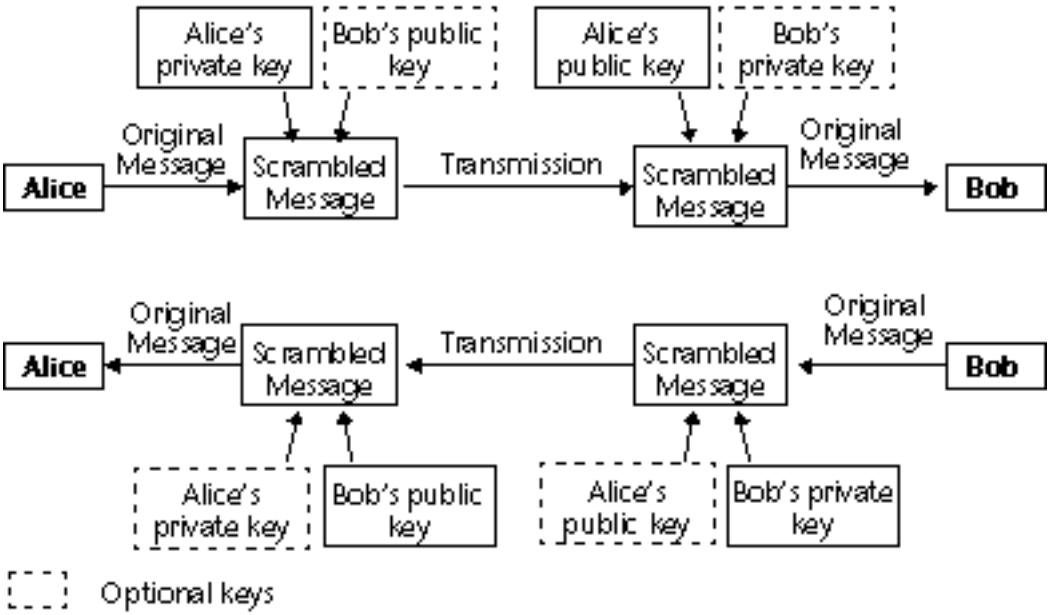
Note: This command uses the SSL protocol algorithm and encryption features. To be able to use this command, make sure that the components necessary to the SSL protocol are installed properly on your machine — even though you do not want to use SSL for 4D Web server connections. For detailed information on this protocol, please refer to section Web Services, Using SSL 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.

Examples

• 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
```

See Also

DECRYPT BLOB, GENERATE ENCRYPTION KEYPAIR.

version 6.5.3 (Modified)

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.

Examples

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
```

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
```

See Also

BLOB PROPERTIES, COMPRESS BLOB.

System Variables or Sets

The OK variable is set to 1 if the BLOB has been successfully expanded, otherwise it is set to 0.

INSERT IN BLOB

version 6.0

INSERT IN BLOB (blob; offset; len{; filler})

Parameter	Type		Description
blob	BLOB	→	BLOB into which bytes will be inserted
offset	Variable	→	Starting position where bytes will be inserted
len	Number	→	Number of bytes to be inserted
filler	Number	→	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 variable parameter the position of the insertion relative to the beginning of the BLOB.

See Also

DELETE FROM BLOB.

INTEGER TO BLOB (integer; blob; byteOrder{; offset | *})

Parameter	Type		Description
integer	Number	→	Integer value to write into the BLOB
blob	BLOB	→	BLOB to receive the Integer value
byteOrder	Number	→	0 Native byte ordering 1 Macintosh byte ordering 2 PC byte ordering
offset | *	Variable | *	←	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
Native byte ordering	Long Integer	0
Macintosh byte ordering	Long Integer	1
PC byte ordering	Long Integer	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.

Examples

1. After executing this code:

```
INTEGER TO BLOB (0x0206;vxBlob;Native byte ordering)
```

- The size of vxBlob is 2 bytes
- On Macintosh vxBLOB{0} = $02 and vxBLOB{1} = $06
- On PC vxBLOB{0} = $06 and vxBLOB{1} = $02

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

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

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

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)

See Also

BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, LONGINT TO BLOB, REAL TO BLOB, TEXT 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
*	*	→	* 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.

See Also

BLOB to list, BLOB TO VARIABLE, SAVE LIST, VARIABLE TO BLOB.

LONGINT TO BLOB

version 6.0

LONGINT TO BLOB (longInt; blob; byteOrder{; offset | *})

Parameter	Type		Description
longInt	Number	→	Long Integer value to write into the BLOB
blob	BLOB	→	BLOB to receive the Long Integer value
byteOrder	Number	→	0 Native byte ordering 1 Macintosh byte ordering 2 PC byte ordering
offset | *	Variable | *	→	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
Native byte ordering	Long Integer	0
Macintosh byte ordering	Long Integer	1
PC byte ordering	Long Integer	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.

Examples

1. After executing this code:

```
LONGINT TO BLOB (0x01020304;vxBlob;Native byte ordering)
```

- The size of vxBlob is 4 bytes
- On Macintosh vxBLOB{0}=$01, vxBLOB{1}=$02, vxBLOB{2}=$03, vxBLOB{3}=$04
- On PC vxBLOB{0}=$04, vxBLOB{1}=$03, vxBLOB{2}=$02, vxBLOB{3}=$01

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

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

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

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)

See Also

BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, INTEGER TO BLOB, REAL TO BLOB, TEXT TO BLOB.

REAL TO BLOB (real; blob; realFormat{; offset | *})

Parameter	Type		Description
real	Number	→	Real value to write into the BLOB
blob	BLOB	→	BLOB to receive the Real value
realFormat	Number	→	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) 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
Native real format	Long Integer	0
Extended real format	Long Integer	1
Macintosh double real format	Long Integer	2
PC double real format	Long Integer	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.

Examples

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

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

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

4. After executing this code:

```
SET BLOB SIZE (vxBlob;100)
C_REAL (vrValue)
vrValue := ...
INTEGER TO BLOB (vrValue;vxBlob;Windows Double real format)
    ` or Macintosh double real format
```

• On all platforms, the size of vxBlob is 8 bytes

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

6. After executing this code:

```
SET BLOB SIZE (vxBlob;100)
C_REAL (vrValue)
vrValue := ...
vlOffset:=50
REAL TO BLOB (vrValue;vxBlob;Windows 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)

See Also

BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, TEXT TO BLOB.

SET BLOB SIZE

version 6.0

SET BLOB SIZE (blob; size{; filler})

Parameter	Type		Description
blob	BLOB	→	BLOB field or variable
size	Number	→	New size of the BLOB
filler	Number	→	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.

Examples

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)
```

2. The following example creates a BLOB of 16K filled of 0xFF:

```
C_BLOB(vxData)
SET BLOB SIZE(vxData;16*1024;0xFF)
```

See Also

BLOB size.

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.

version 11 (Modified)

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	Number	→	Format and character set of text
offset | *	Variable | *	→	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 fixes the internal format and the character set of the text value to be written. 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	Long Integer	0
Mac Pascal string	Long Integer	1
Mac Text with length	Long Integer	2
Mac Text without length	Long Integer	3
UTF8 C string	Long Integer	4
UTF8 Text with length	Long Integer	5
UTF8 Text without length	Long Integer	6

Notes:

• The "UTF8" constants can only be used when the application runs in Unicode mode.
• The Mac Text with length constant 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 3-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
```

See Also

BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, CONVERT FROM TEXT, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB.

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 | *	Character	→	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
- Two-dimensional arrays

However, if you store a Long Integer variable that is a reference to a hierarchical list (ListRef), VARIABLE TO BLOB will store the Long Integer variable, not the list. To store and retrieve hierarchical lists in and from a BLOB, use the commands LIST TO BLOB and BLOB to list.

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.

Examples

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)
```

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 (${$vlParam}->;$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 ($1->;${$vlParam}->;$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)
```

See Also

BLOB to list, BLOB TO VARIABLE, LIST TO BLOB.

System Variables or Sets

The OK variable is set to 1 if the variable has been successfully stored, otherwise it is set to 0.

7

Boolean

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)
```

See Also

False, Logical Operators, Not, True.

In addition, the following 4D commands return a Boolean result: Activated, After, Before, Before selection, Before subselection, Caps lock down, Is compiled mode, Deactivated, During, End selection, End subselection, In break, In footer, In header, In transaction, Is a list, Is a variable, Is in set, Is user deleted, Locked, Macintosh command down, Macintosh control down, Macintosh option down, Modified, Modified record, Nil, Outside call, Read only state, Semaphore, Shift down, True, Undefined, User in group, Windows Alt down, Windows Ctrl down.

True

Boolean

version 3

True → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	True

Description

True returns the Boolean value True.

Example

The following example sets the variable vbOptions to True:

```
vbOptions:=True
```

See Also

False, Not.

False

Boolean

version 3

False → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	False

Description

False returns the Boolean value False.

Example

The following example sets the variable vbOptions to False:

```
vbOptions:=False
```

See Also

Not, True.

Not (boolean) → Boolean

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
```

8

Communications

GET SERIAL PORT MAPPING (numArray; nameArray)

Parameter	Type		Description
numArray	Number 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
    `params contains the communication parameters
SET CHANNEL($vFinalPortNum;params)
...  `Carry out the desired operations
SET CHANNEL(11)    `Closing of port
```

See Also

SET CHANNEL.

RECEIVE BUFFER (receiveVar)

Parameter	Type		Description
receiveVar	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 9.x serial port buffer is limited in size to 10 Kbytes. Under Mac OS X, its 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.

Note: There are 4D plug-ins that enable you to increase the size of the serial buffer.

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.

See Also

ON ERR CALL, RECEIVE PACKET, Semaphore, SET CHANNEL, Variables.

RECEIVE PACKET ({docRef; }receiveVar; stopChar | numChars)

Parameter	Type		Description
docRef	DocRef	→	Document reference number, or Current channel (serial port or document)
receiveVar	String var | BLOB var	→	Variable to receive data
stopChar | numChars	String | Number	→	Character(s) at which to stop receiving, or Number of characters to receive

Description

RECEIVE PACKET reads characters from a serial port or from a document.

If docRef is specified, this command reads characters from a document opened using Open document, Create document or Append document. If docRef is omitted, this command reads characters 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 corresond to that of the packet sent.

Notes:
•In non-Unicode mode (compatibility mode), String variables accept up to 255 characters and have a fixed size and Text variables do not have a set size and can accept up to 32,000 characters.
•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.

To read a particular number of characters, pass this number in numChars. If the receiveVar variable is of the Text type, in a single call you can read up to 2 GB of text in Unicode mode or 32,000 characters in non-Unicode mode (in this case, to specify the maximum number of characters, you can pass the MAXTEXTLENBEFOREV11 constant in numChars.

To read characters 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 character 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 and 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.

Note: In non-Unicode mode (compatibility mode), when you use the RECEIVE PACKET command to read characters from a Windows document and do not want to use ASCII maps to convert Windows characters into Mac OS characters, you can use the Win to Mac function.

Examples

1. The following example reads 20 characters from a serial port into the variable getTwenty:

```
RECEIVE PACKET (getTwenty; 20)
```

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))
```

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>")
```

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
```

See Also

Get document position, RECEIVE PACKET, SEND PACKET, SET DOCUMENT POSITION, SET TIMEOUT.

System Variables or 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 {(table)}

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 all subrecords, pictures and BLOBs stored in 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.

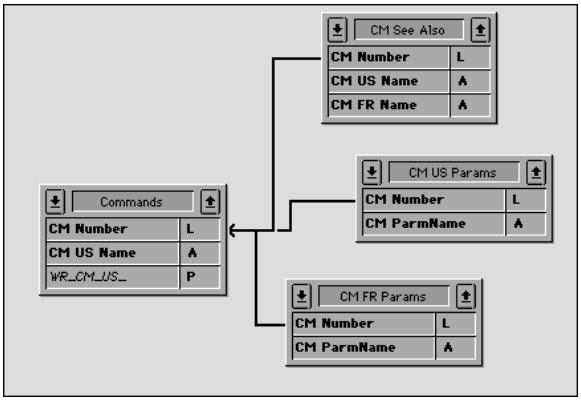
Notes

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, Append document or Create 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, subtables and/or related tables, using SEND RECORD and RECEIVE RECORD is far more convenient.

For instance, the documentation you are currently reading has been created using 4D and 4D Write. Because several writers in different locations wordwide were working on it, we needed 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 RECEIVE VARIABLE 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") | ($CurLang="FR"))  ` Did we get a valid language
                ` How many commands are there in this document?
            RECEIVE VARIABLE($vlNbCmd)
            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.

See Also

RECEIVE VARIABLE, SEND RECORD, SEND VARIABLE.

System Variables or 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 (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, Append document or Create 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 new BLOB commands introduced in version 6.
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.

See Also

ON ERR CALL, RECEIVE RECORD, SEND RECORD, SEND VARIABLE.

System Variables or 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 ({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 Append document. 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 and 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.

Important: In non-Unicode mode (compatibility mode), SEND PACKET writes Mac OS ASCII data on both Windows and Macintosh platforms. Mac OS ASCII data uses eight bits. Standard ASCII uses only the lower seven bits. Many devices do not use the eighth bit in the same way as does Windows/Macintosh. If the string to be sent contains data that uses the eighth bit, be sure to create an ASCII map to translate the ASCII characters, and execute USE CHARACTER SET before using SEND PACKET. You can also use the Mac to Win function (for more information, refer to the example for this function). Protocols like XON/XOFF use some low ASCII codes to establish communication between machines. Be careful to not send such ASCII codes, as this may interfere with the protocol or even break communication.

Examples

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
```

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("âzértÿ";$send_blob;UTF8 Text without length)
SET BLOB SIZE($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
$vlDocRef:=Open document(document)
    End(OK=1)
    RECEIVE PACKET($vlDocRef;$receive_blob;65536)
    CLOSE DOCUMENT($vlDocRef)
End if
```

See Also

Get document position, RECEIVE PACKET, SET DOCUMENT POSITION.

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 all subrecords, pictures and BLOBs stored in 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, Append document or Create document.

Example

See example for the command RECEIVE RECORD.

See Also

RECEIVE RECORD, RECEIVE VARIABLE, 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, Append document or Create 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 new BLOB commands introduced in version 6.

Example

See example for the command RECEIVE RECORD.

See Also

RECEIVE RECORD, RECEIVE VARIABLE, SEND RECORD, SET CHANNEL.

SET CHANNEL (port | operation{; settings | document})

Parameter	Type		Description
port | operation	Number	→	Serial port number, or Document operation to perform
settings | document	Number | String	→	Serial port settings, or 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 SEND PACKET 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 RECORD or RECEIVE VARIABLE.

The port Parameter

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.

Examples

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)
```

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)

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

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 (in baud)	380	300
	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. 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. 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.

Examples

See examples for the commands RECEIVE BUFFER, SET TIMEOUT and RECEIVE RECORD.

See Also

Append document, Create document, GET SERIAL PORT MAPPING, Open document, RECEIVE BUFFER, RECEIVE PACKET, RECEIVE RECORD, RECEIVE VARIABLE, SEND PACKET, SEND RECORD, SEND VARIABLE, SET TIMEOUT.

SET TIMEOUT (seconds)

Parameter	Type		Description
seconds	Number	→	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("")
```

See Also

ON ERR CALL, RECEIVE BUFFER, RECEIVE PACKET, RECEIVE RECORD, RECEIVE VARIABLE.

USE CHARACTER SET (map | *{; mapInOut})

Parameter	Type		Description
map | *	String | *	→	Name of character set to use (Unicode mode), or Document name of ASCII map to use (ASCII mode) or * to reset to default character set/ASCII map
mapInOut	Number	→	0 = Output map 1 = Input map If omitted, output map

Description

USE CHARACTER SET can be used to modify the character set used by 4D during transfer of data between the database and a document or a serial port. Transfer operations include the import and export of text (ASCII), DIF, and SYLK files. An ASCII 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 and RECEIVE PACKET (for BLOB type packets) and RECEIVE VARIABLE.

The USE CHARACTER SET command is used differently according to whether the database is operating in Unicode mode or in ASCII compatibility mode. It loads into memory either a character set or an ASCII map.

Note: For more information about these modes, please refer to the About Unicode section.

Unicode Mode
In Unicode mode, 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.

ASCII compatibility mode
In this mode, the command loads into memory and uses the ASCII map document (passed in map) that was previously saved. The ASCII map must have been created beforehand using a previous version of 4D. If you give an empty string for map, USE CHARACTER SET displays a standard Open File dialog box so that the user can specify an existing ASCII map.

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 Unicode mode in 4D v11, the default character set is UTF-8.
In compatibility mode compatibilité, the standard Mac ASCII is restored.

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
```

See Also

EXPORT DIF, EXPORT SYLK, EXPORT TEXT, IMPORT DIF, IMPORT SYLK, IMPORT TEXT, Mac to Win, RECEIVE BUFFER, RECEIVE PACKET, SEND PACKET, Win to Mac.

System Variables or Sets

The OK system variable is set to 1 if the map is loaded correctly. Otherwise, it is set to 0.

9

Compiler

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.

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_INTEGER	C_REAL	IDLE
C_BOOLEAN	C_LONGINT	C_STRING	
C_DATE	C_PICTURE	C_TEXT	
C_GRAPH	C_POINTER	C_TIME	

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).

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.

Examples

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
    ` The interprocess variable <>OnWindows is declared as a variable of type Boolean
C_BOOLEAN(<>OnWindows)
    ` The local variable $vdCurDate is declared as a variable of type Date
C_DATE($vdCurDate)
    ` The 3 process variables vg1, vg2 and vg3 are declared as variables of type Graph
C_GRAPH(vg1;vg2;vg3)
```

2. In the following example, the project method OneMethodAmongOthers declares 3 parameters:

```
    ` OneMethodAmongOthers Project Method
    ` OneMethodAmongOthers ( Real ; Integer { ; Long } )
    ` OneMethodAmongOthers ( Amount ; Percentage { ; Ratio } )

C_REAL($1)      ` 1st parameter is of type Real
C_INTEGER($2)      ` 2nd parameter is of type Integer
C_LONGINT($3)          ` 3rd parameter is of type Long Integer

` ...
```

3. In the following example, the project method Capitalize accepts a string parameter and returns a string result:

```
    ` Capitalize Project Method
    ` Capitalize ( String ) -> String
    ` Capitalize ( Source string ) -> Capitalized string

C_STRING(255;$0;$1)
$0:=Uppercase(Substring($1;1;1))+Lowercase(Substring($1;2))
```

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
```

5. In the following example, the project method COMPILER_Param_Predeclare28 predeclares the syntax of other project methods for the compiler:

```
    ` COMPILER_Param_Predeclare28 Project Method

    ` OneMethodAmongOthers ( Real ; Integer { ; Long } )
C_REAL(OneMethodAmongOthers;$1)
C_INTEGER(OneMethodAmongOthers;$2)        ` ...
C_LONGINT(OneMethodAmongOthers;$3)        ` ...
C_STRING(Capitalize;255;$0;$1)    ` Capitalize ( String ) -> String
C_TIME(SEND PACKETS;$1)` SEND PACKETS ( Time ; Text { ; Text2... ; TextN } )
C_TEXT(SEND PACKETS;${2})   ` ...
```

See Also

C_BLOB, C_BOOLEAN, C_DATE, C_GRAPH, C_INTEGER, C_LONGINT, C_PICTURE, C_POINTER, C_REAL, C_STRING, C_TEXT, C_TIME, IDLE.

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.

Variable types
All variables have a type. As described in the Data Types section, there are 12 different types of variables:
Boolean
Fixed string
Date
Integer
Longint
Graph
Time
Picture
Number (or Real)
Pointer
Text
BLOB

There are nine different types of arrays:
Boolean Array
String Array
Date Array
Integer Array
Longint Array
Picture Array
Real Array
Pointer Array
Text Array

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.

Typing variables
The compiler must respect the identification criteria of the variables.
There are two possibilities:
• 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.

• 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 C_INTEGER are actually the same as those declared by the directive C_LONGINT. They are, in fact, long integers between –2147483648 and +2147483647.

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.

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.

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
```

The C_STRING compiler directive
The C_STRING command uses a different syntax than the other directives because it accepts an additional parameter—the maximum string length.
C_STRING(length;var1{;var2;…;varN})

Since C_STRING types fixed-length strings, you specify the maximum length of such strings. In a compiled database, you must specify the length of the string with a constant rather than with a variable.
In an interpreted database, the following sequence is acceptable:

```
TheLength:=15
C_STRING(TheLength;TheString)
```

4D interprets *TheLength*, then replaces *TheLength* with its value in the C_STRING compiler directive.
However, the compiler uses this command when typing variables with no specific assignment in mind. Thus, it is not in a position to know that *TheLength* equals 15. Not knowing the string's length, the compiler cannot keep a space for it in the symbol table. Therefore, with compilation in mind, use a constant to specify the length of the declared character string. For example, use a statement such as this:

```
C_STRING(15;TheString)
```

The same rule applies to declaring fixed string arrays, which are typed with the command:

```
ARRAY STRING(length;arrayName;size)
```

The parameter that indicates string lengths in the array must be a constant.

However, you can specify the length of the string with a 4D constant or a hexadecimal value in these two compiler directives. For example:

```
C_STRING(4DConstant;TheString)
ARRAY STRING(4DConstant;TheArray;2)
C_STRING(0x000A;TheString)
ARRAY STRING(0x000A;TheArray;2)
```

Do not confuse the length of an Alphanumeric field, which has a maximum of 80 characters, with a fixed string variable. The maximum length of a string declared by a C_STRING directive, or belonging to an ARRAY STRING, is between 1 and 255.

Note: The syntax of this command allows you to declare several variables of the same length in a single line. If you want to declare several strings of different lengths, do so on separate lines.

A certain liberty permitted by the compiler

Compiler directives remove any ambiguity concerning data types and, in the case of alphanumeric strings, lengths. 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, or if you assign a string of 30 characters to a variable declared as a 10-character string, 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_INTEGER(vInteger)
vInteger:=2.5
```

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 2.5). Similarly, if you declare a string that is shorter than the one you are dealing with, the compiler will only take the number of characters declared in the directives. Therefore, in the following sequence:

```
C_STRING(10;MyString)
MyString:="It is a beautiful day"
```

the compiler takes the first ten characters of the constant, i.e. “It is a be”.

See Also

Error messages, Optimization Hints, Syntax Details, 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_INTEGER(<>vInteger)   `Declaration of an Integer 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.

Keep in mind that a data type conflict can arise concerning the use of C_STRING if you modify the maximum string length. Thus, if you write:

```
C_STRING(5;MyString)
MyString:="Hello"
C_STRING(7;MyString)
MyString:="Flowers"
```

the compiler identifies a conflict because it must provide an adequately-sized location when declaring String variables.
The solution is to use a compiler directive that gives the maximum length, since, by default, the compiler will accept a shorter length. You can write:

```
C_STRING(7;String)
String:="Flowers"
String:="Hello"
```

Note: If you have written C_STRING(7;String) twice, i.e.:

```
C_STRING(7;String)
String:="Flowers"
C_STRING(7;String)
String:="Hello"
```

the compiler will nevertheless accept the directives; the second directive is simply redundant.

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.

Case of fixed string arrays
String arrays follow the same rules as fixed strings, for the same reasons.
If you write:

ARRAY STRING(5;MyArray;10)
ARRAY STRING(10;MyArray;10)

the compiler detects a length conflict. The solution is simple: declare the maximum string length. The compiler automatically handles shorter length strings.

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, the number of dimensions, or, in a String array, the string length. You will thus find yourself in one of the three 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.

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.

Graph variable

A graph area is automatically data type Graph (Longint). This variable never creates a data type conflict. For a Graph type variable, 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.

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.

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.

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 and tab controls).

• 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 STRING, ARRAY TEXT...).

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_INTEGER(${4})
```

This command means that all parameters starting from the fourth (included) will be addressed by indirection and will be of the data type Integer. $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 Integer, 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	String (255)
FldDelimit	Longint
RecDelimit	Longint
Error	Longint
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 in this manual. 4D constants are also displayed in the Explorer, in Design mode.

See Also

Error messages, Optimization Hints, Syntax Details, Using Compiler Directives.

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.
The two functions are not sufficient for the compiler to determine the data type of the result. In such cases, use a compiler directive to avoid any ambiguity.

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 and 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.
These commands do not require that the array passed as a parameter be declared. By default, the non-declared array will be typed as a Text array. If the array was previously declared as String or Text, these commands will follow your directives.

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, this 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("INPUT FORM (Form"+String(i)+")")
```

It can be replaced by:

```
i:=FormFunc
VarForm:="Form"+String(i)
INPUT FORM(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

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
APPLY TO SELECTION
COPY NAMED SELECTION
CREATE EMPTY SET
CREATE SET
CUT NAMED SELECTION
DIALOG
EXPORT DIF
EXPORT SYLK
EXPORT TEXT
GOTO RECORD
GOTO SELECTED RECORD
GRAPH TABLE
FORMULA
IMPORT DIF
IMPORT SYLK
IMPORT TEXT
INPUT FORM
LOAD SET
LOCKED ATTRIBUTES
ORDER BY
ORDER BY FORMULA
OUTPUT FORM
PAGE SETUP
Print form
PRINT LABEL
QR REPORT
QUERY
QUERY BY FORMULA
QUERY SELECTION
QUERY SELECTION BY
REDUCE SELECTION
RELATE MANY
REMOVE FROM SET

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.

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.

See Also

Error messages, Optimization Hints, Typing Guide, Using Compiler Directives.

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 Preferences 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 directives C_INTEGER or 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...). 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

The default type assigned to alphanumeric variables is Text if the Preferences are not set to anything else. For example, if you write:
MyString:="Hello", MyString would be typed as a Text variable by the compiler.
If this variable will be processed frequently, it is worthwhile to declare it using C_STRING. Processing is much faster with String type variables, which have a defined maximum length, than with Text variables. Keep in mind the rules governing the behavior of this directive.

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.

See Also

Error messages, Syntax Details, Typing Guide, Using Compiler Directives.

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)
```

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.
The messages are grouped by the following topics: Typing, Syntax, Parameters, Operators, Plug-in Commands and General Errors.

• 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 maximum length of a string.
```
C_STRING(3;MyString)
C_STRING(5;MyString)
```

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;"")
```

Constant number expected.

```
C_STRING(Variable;MyString)
```

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 length of a string cannot be greater than 255 characters.

```
C_STRING(325;MyString)
```

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:
Too many opening character reference symbols.

```
MyString[[3:="a"
```

Too many closing character reference symbols.

```
MyString3]]:="a"
```

• Windows:
Too many opening character reference symbols.

```
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.

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 Formula 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.

See Also

Optimization Hints, Syntax Details, Typing Guide, Using Compiler Directives.

C_BLOB ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	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 section Compiler Commands.

See Also

Compiler Commands.

C_BOOLEAN ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	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.

See Also

Compiler Commands, Count parameters.

C_DATE ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	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.

See Also

Compiler Commands, Count parameters.

C_GRAPH ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	String	→	Name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The C_GRAPH command casts each specified variable as a Graph 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_GRAPH(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_GRAPH(${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.

See Also

Compiler Commands.

C_INTEGER ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Note: This command is still present in 4D for compatibility with old databases. In fact, 4D and the compiler retype Integers into Longints internally. For example :
C_INTEGER($MyVar)
$TheType:=**Type**($MyVar) `$TheType = 9 (Is Longint)

Description

The C_INTEGER command casts each specified variable as an 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_INTEGER(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_INTEGER(${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.

See Also

Compiler Commands, Count parameters, C_LONGINT, C_REAL.

C_LONGINT ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	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.

See Also

Compiler Commands, Count parameters, C_INTEGER, C_REAL.

C_PICTURE ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	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.

See Also

Compiler Commands, Count parameters.

C_POINTER ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	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.

See Also

Compiler Commands, Count parameters.

C_REAL ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	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.

See Also

Compiler Commands, Count parameters, C_INTEGER, C_LONGINT.

C_STRING ({method; }size; variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
size	Number	→	Size of the string
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The C_STRING command casts each specified variable as a String variable.

The size parameter specifies the maximum length of the strings that the variable can contain. Strings are limited to 255 characters. If speed is a concern, use string variables rather than text variables wherever possible.

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_STRING(...;${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_STRING(...;${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.

See Also

Compiler Commands, Count parameters, C_TEXT.

C_TEXT ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	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.

See Also

Compiler Commands, Count parameters, C_STRING.

C_TIME ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	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.

See Also

Compiler Commands, Count parameters.

IDLE

Parameter	Type	Description

This command 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
```

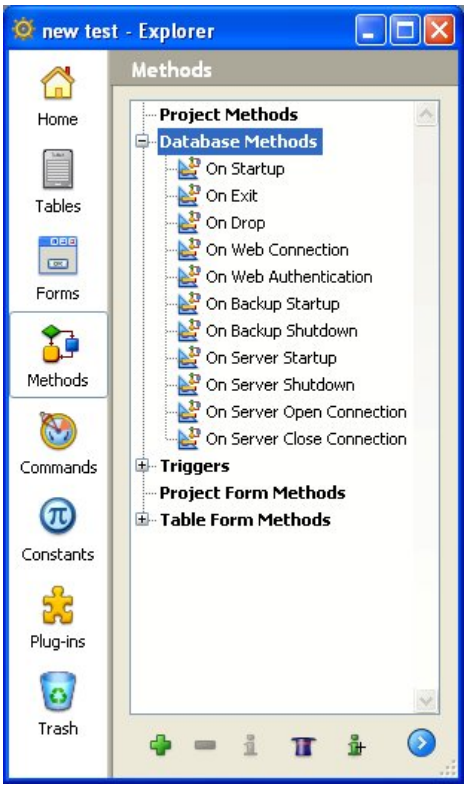
See Also

Compiler Commands, ON EVENT CALL.

10

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 execution of database methods:

Database Method	4D Developer	4D Server	4D Client
On Startup	Yes, Once	No	Yes, Once
On Exit	Yes, Once	No	Yes, Once
On Drop	Yes, Multiple	No	Yes, Multiple
On Web Connection	Yes, Multiple	Yes, Multiple	No
On Web Authentication	Yes, Multiple	Yes, Multiple	No
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

For detailed information about each of the database methods, see the following sections:

- On Startup Database Method
- On Exit Database Method
- On Drop Database Method
- On Web Connection Database Method
- On Web Authentication Database Method
- On Backup Startup Database Method
- On Backup Shutdown Database Method
- On Server Startup Database Method (*4D Server Reference* manual)
- On Server Shutdown Database Method (*4D Server Reference* manual)
- On Server Open Connection Database Method (*4D Server Reference* manual)
- On Server Close Connection Database Method (*4D Server Reference* manual)

The On Exit Database Method is called once when you quit a database.

This method is used in the following 4D environments:
• 4D Developer
• 4D Client (on the client side)
• 4D Desktop and 4D Interpreted Desktop
• 4D application compiled and merged with 4D Unlimited Desktop

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.

See Also

On Startup Database Method, QUIT 4D.

On Startup Database Method

version 11 (Modified)

The On Startup Database Method is called once when you open a database.

This occurs in the following 4D environments:
• 4D Developer
• 4D Client (on the client side, after the connection has been accepted by 4D Server)
• 4D Desktop and 4D Interpreted Desktop
• 4D application compiled and merged with 4D Unlimited 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.

Example
See the example in the section On Exit Database Method.

See Also

Database Methods, Methods, On Exit Database Method, QUIT 4D.

11

Data Entry

ADD RECORD ({aTable}{; }{*})

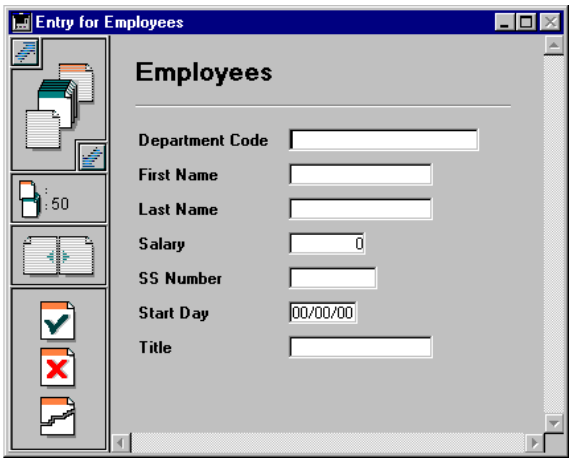
Parameter	Type		Description
aTable	Table	→	Table to use for data entry, or Default table, if omitted
*		→	Hide scroll bars

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 or a size box.

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.

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.

Examples

1. The following example is a loop commonly used to add new records to a database:

```
INPUT FORM ([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
```

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])
INPUT FORM([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
```

See Also

ACCEPT, CANCEL, CREATE RECORD, MODIFY RECORD, SAVE RECORD.

System Variables or 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.

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.

Description

The ADD SUBRECORD command lets the user add a new subrecord to subtable, using the form form. ADD SUBRECORD creates a new subrecord in memory, makes it the current subrecord, and displays form. A current record for the parent table must exist. If a current parent record does not exist for the process, ADD SUBRECORD has no effect. The form must belong to subtable.

The subrecord is kept in memory (accepted) if the user clicks an Accept button or presses the Enter key (numeric pad), or if the ACCEPT command is executed. After the subrecord has been added, the parent record must be explicitly saved in order for the subrecord to be saved.

The subrecord is not saved 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.

After a call to ADD SUBRECORD, OK is set to 1 if the subrecord is accepted, to 0 if canceled.

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.

Example

The following example is part of a method. It adds a subrecord for a new child to an employee's record. The data for the children is stored in a subtable named [Employees]Children. Note that the [Employees] record must be saved in order for the new subrecord to be saved:

```
ADD SUBRECORD([Employees]Children;"Add Child")
If (OK=1)  ` If the user accepted the subrecord
   SAVE RECORD ([Employees])  ` save the employee's record
End if
```

See Also

ACCEPT, CANCEL, CREATE SUBRECORD, DELETE SUBRECORD, MODIFY SUBRECORD, SAVE RECORD.

System Variables or Sets

Accepting the subrecord sets the OK system variable to 1; canceling it sets the OK system variable to 0.

DIALOG ({aTable; }form; *)

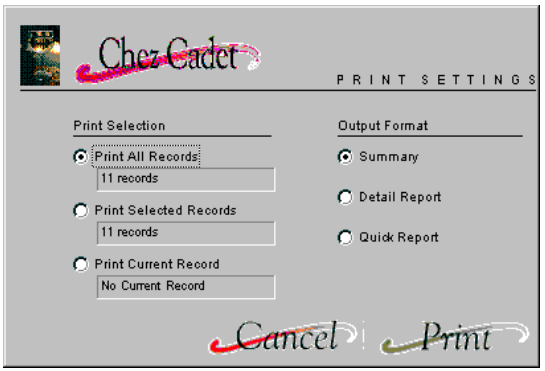
Parameter	Type		Description
aTable	Table	→	Table owning the form or Default table if omitted
form	Form	→	Form to display as dialog
*	*	→	Use the same process

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: 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 combination (Ctrl-Period on Windows, Command-Period 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 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.

After a call to DIALOG, if the dialog is accepted, OK is set to 1; if it is canceled, OK is set to 0.

Examples

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
```

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";Standard form window)
DIALOG("doc")
```

See Also

ACCEPT, ADD RECORD, CANCEL, Open window.

System Variables or 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.

version 2004 (Modified)

Compatibility note
This function is kept for compatibility reasons only since it is based on the former management of execution cycles, which is itself obsolete since version 6 (see the Before, During, etc. commands). In most cases, it is now strongly recommended to use the Form event command and to check whether it returns the On Data Change event.

Modified (field) → Boolean

Parameter	Type		Description
field	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 During execution cycle.

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 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.

Examples

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.

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
```

See Also

Form event, Old.

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 lets the user 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 command ADD RECORD.

See Also

ADD RECORD, Locked, Modified record, READ WRITE, UNLOAD RECORD.

System Variables or 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.

MODIFY SUBRECORD (subtable; form{; *})

Parameter	Type		Description
subtable	Subtable	→	Subtable to use for data entry
form		→	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.

Description

The MODIFY SUBRECORD command displays the current subrecord of subtable for modification using the form form. The form must belong to subtable.

A current record for the parent table must exist. If a current parent record does not exist for the process, MODIFY SUBRECORD has no effect. In addition, if there is no current subrecord, then MODIFY SUBRECORD does nothing.

Any modifications made to the subrecord will actually only be saved if the parent record itself is saved.

After a call to MODIFY SUBRECORD, OK is set to 1 if the subrecord modifications are accepted, to 0 if canceled.

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.

See Also

ACCEPT, ADD SUBRECORD, CANCEL, SAVE RECORD.

System Variables or Sets

Accepting the subrecord modifications sets the OK system variable to 1; canceling it sets the OK system variable to 0.

Old (aField) → Expression

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.

See Also

Modified.

12

Date and Time

Add to date (aDate; years; months; days) → Date

Parameter	Type		Description
aDate	Date	→	Date to which to add days, months, and years
years	Number	→	Number of years to add to the date
months	Number	→	Number of months to add to the date
days	Number	→	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)
```

See Also

Date Operators.

Current date {(*)} → Date

Parameter	Type		Description
*		→	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.

Examples

1. The following example displays an alert box containing the current date:

```
ALERT("The date is " + String(Current date)+".")
```

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
    ` Eliminate the first digits as well as the first / separator
$vsDate:=Substring($vsDate;$vlPos+1)
Case of
    : ($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...
```

See Also

Date Operators, Day of, Month of, Year of.

Current time {(*)} → Time

Parameter	Type		Description
*		→	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.

Examples

1. The following example shows you how to time the length of an operation. Here, LongOperation is a method that needs to be timed:

```
$vhStartTime:=Current time  ` Save the start time
LongOperation  ` Perform the operation
   ` Display how long it took
ALERT ("The operation took "+String(Current time–$vhStartTime))
```

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))
```

See Also

Milliseconds, String, Tickcount, Time Operators.

Date (dateString) → Date

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 must follow the normal rules for the date format.

In the US version of 4D, 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 adds 19 to the beginning of the year, unless you have change this default using the command SET DEFAULT CENTURY. The following characters are valid date separators: slash (/), space, period (.), and comma (,).

Date does not check whether or not dateString is a valid date. If an invalid date (such as "13/35/94") is passed, Date will return the invalid date. However, if dateString could not possibly be interpreted as a date (for example, "aa/12/94"), the null date value (!00/00/00!) is returned.

It is your responsibility to verify that dateString is a valid date.

Examples

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
```

2. The following example returns the string "12/12/94" as a date:

```
vdDate:=Date("12/12/94")
```

Day number (aDate) → Number

Parameter	Type		Description
aDate	Date	→	Date for which to return the number
Function result	Number	←	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:

Constants	Type	Value
Monday	Long Integer	2
Tuesday	Long Integer	3
Wednesday	Long Integer	4
Thursday	Long Integer	5
Friday	Long Integer	6
Saturday	Long Integer	7
Sunday	Long Integer	1

Note: Day number of 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
```

See Also

Day of.

Day of (aDate) → Number

Parameter	Type		Description
aDate	Date	→	Date for which to return the day
Function result	Number	←	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.

Examples

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
```

2. See the example for the command Current date.

See Also

Day number, Month of, Year of.

Milliseconds → Longint

Parameter	Type		Description
This command does not require any parameters			
Function result	Longint	←	Number of milliseconds elasped since the machine was started

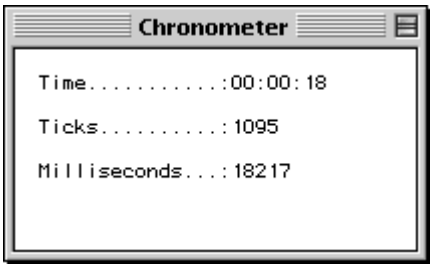
Description

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
```



See Also

Current time, Tickcount.

Month of (aDate) → Number

Parameter	Type		Description
aDate	Date	→	Date for which to return the month
Function result	Number	←	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	Long Integer	1
February	Long Integer	2
March	Long Integer	3
April	Long Integer	4
May	Long Integer	5
June	Long Integer	6
July	Long Integer	7
August	Long Integer	8
September	Long Integer	9
October	Long Integer	10
November	Long Integer	11
December	Long Integer	12

Examples

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
```

2. See example for the command Current date.

See Also

Day of, Year of.

SET DEFAULT CENTURY (century{; pivotYear})

Parameter	Type		Description
century	Number	→	Default century (minus one) for entry of date with two-digit year
pivotYear	Number	→	Pivot year for entry of date with two-digit year

Description

The SET DEFAULT CENTURY command allows you to specify 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 SET DEFAULT CENTURY 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:

```
` Switch to 21st century for default century if year is less than 95
SET DEFAULT CENTURY(19;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 → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	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.

See Also

Current time, Milliseconds.

Time (timeString) → Time

Parameter	Type		Description
timeString	Time	→	Time for which to return number of seconds
Function result	Time	←	Time specified by timeString

Description

The Time command returns a time expression equivalent to the time specified as a string by timeString.

The timeString parameter must contain 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).

Example

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))
```

See Also

String, Time string.

Time string (seconds) → String

Parameter	Type		Description
seconds	Number	→	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))
```

See Also

String, Time.

Year of (aDate) → Number

Parameter	Type		Description
aDate	Date	→	Date for which to return the year
Function result	Number	←	Number indicating the year of date

Description

The Year of command returns the year of aDate.

Examples

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
```

2. See example for the command Current date.

See Also

Day of, Month of.

13

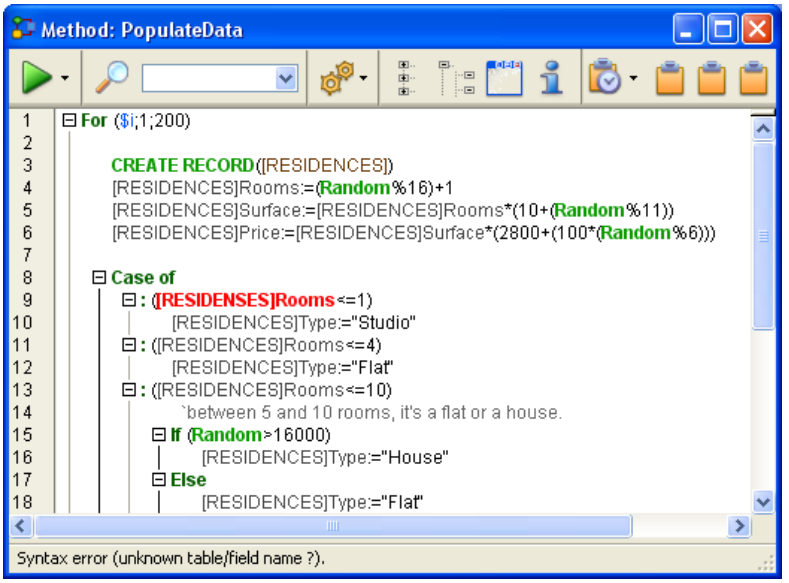
Debugging

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:



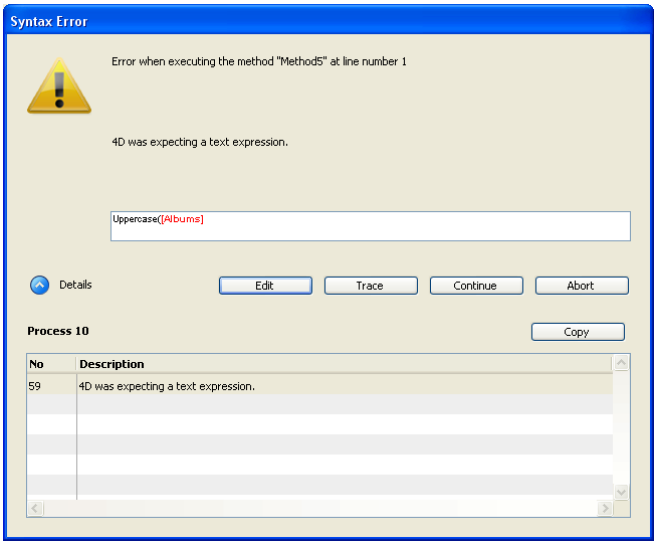
Note: The comments have been manually inserted for the purpose of this manual. Only the color is modified by 4D at the location of the error.

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*.

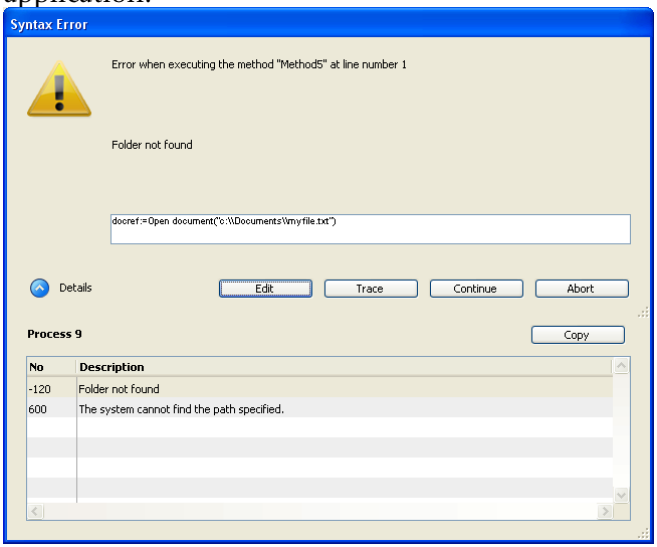
Syntax or Environmental Errors

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 (see the description of 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:

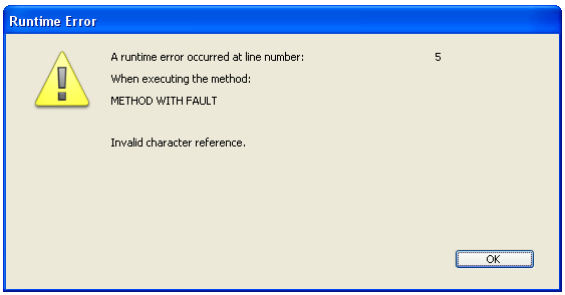
- A syntax error may occur because you try to use a variable that has not yet been initialized.
- An environmental error may occur because you try to open a document whose name is received by a subroutine which does not get the right value in the parameter. Note that in this example, the piece of code 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 unexpecting 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 somethimes pragmatism pays off more quickly. Anyway... use the Debugger.

General Conclusion
Use the Debugger.

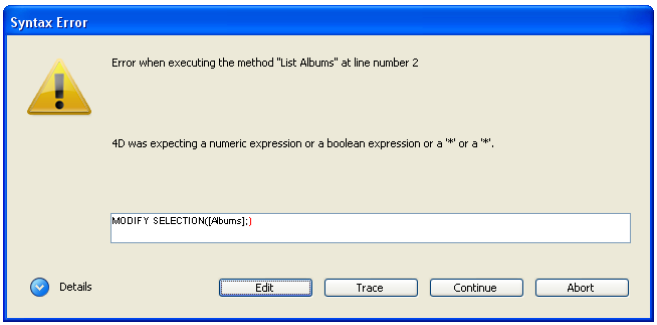
See Also

Break List, Catching Commands, Debugger, Debugger Shortcuts, ON ERR CALL, Syntax Error Window.

The Syntax Error Window is displayed when method execution is halted. Method execution can be halted for either of two reasons:

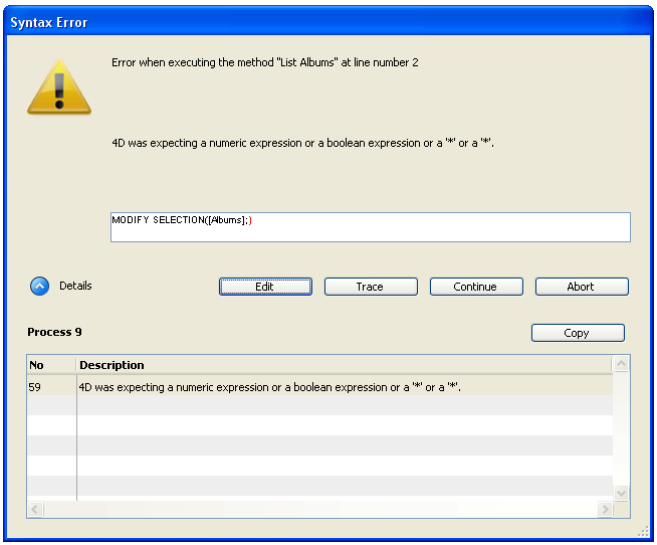
- 4D halts execution because there is an error preventing further method execution.
- You generate a user interrupt by pressing Alt+Click (Windows) or Option+Click (Macintosh) while a method is executing.

The Syntax Error window is shown here:



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.

• **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.

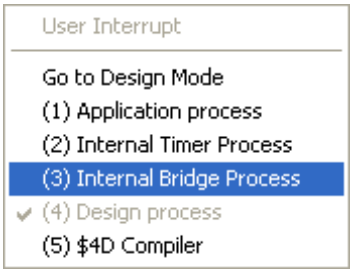
See Also

Debugger, ON ERR CALL, Why a 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:

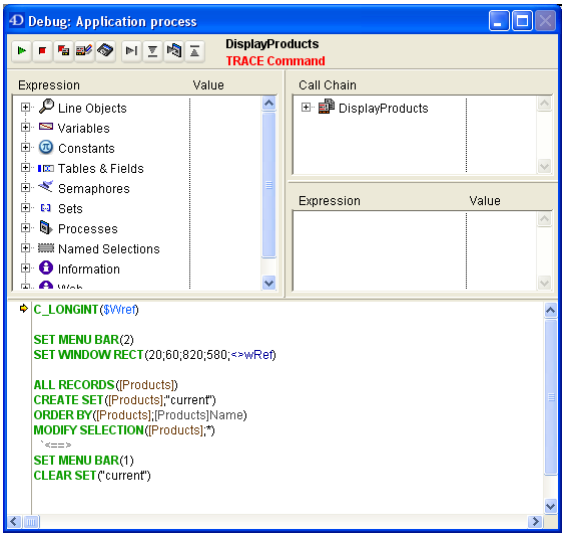
- Clicking the **Trace** button in the Syntax Error Window
- Using the TRACE command
- Clicking the **Debug** button in the Execute Method window.
- Pressing Alt+Shift+Right click (Windows) or Control+Option+Command+Click (Macintosh) while a method is executing, then selecting the process to trace in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

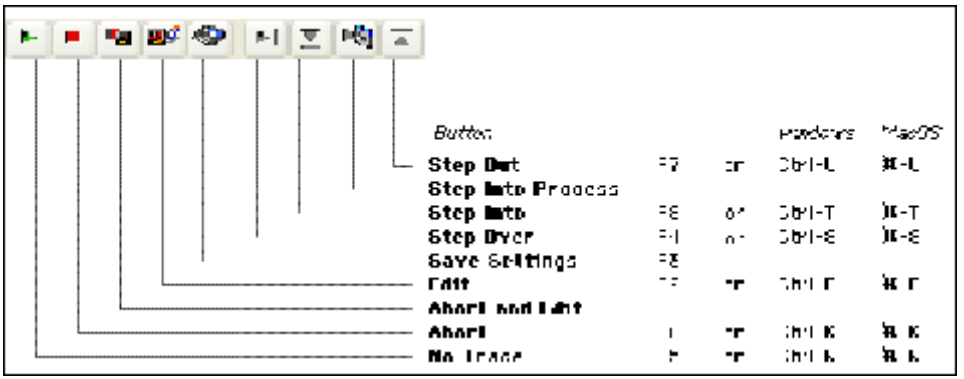


You can move the Debugger Window and/or resize any of its internal window panes as necessary. Displaying a new debug window uses the same configuration (size and position of the window, placing of the division lines and contents of the area that evaluates the expressions) as the last window displayed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

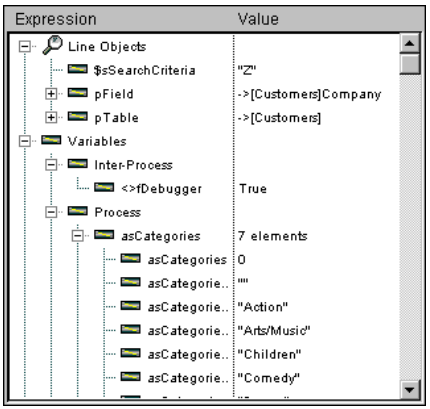
- Watch Pane
- Call Chain Pane
- Custom Watch Pane
- Source Code Pane

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.

See Also

Break List, Call Chain Pane, Catching Commands, Custom Watch Pane, Debugger Shortcuts, ON ERR CALL, Source Code Pane, Syntax Error Window, TRACE, Watch Pane, Why a Debugger?.

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
- Constants
- Fields
- Semaphores
- Sets
- Processes
- Named Selections
- Information
- Cache Statistics

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.

Cache Statistics: Displays statistics regarding the use of tables, index pages, and named selections that are loaded in 4D's cache. The expressions from this theme cannot be modified.

Information: Displays general information, such the current Default Table (if any). The expressions from this theme cannot be modified.

Named Selections: 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.

Processes: 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.

Sets: 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.

Semaphores: 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.

Tables & 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.

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

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 modifiy 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 modifiy 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.

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
```

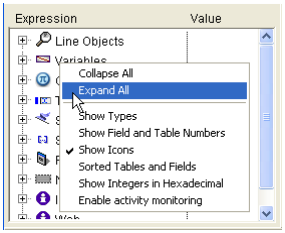
The c variable is shown because it is used in the line to be executed (but has not yet been initialized). The a and b variables are shown because there were used in the previous line and are used in the line to be executed. And so on...

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.

Contextual Menu

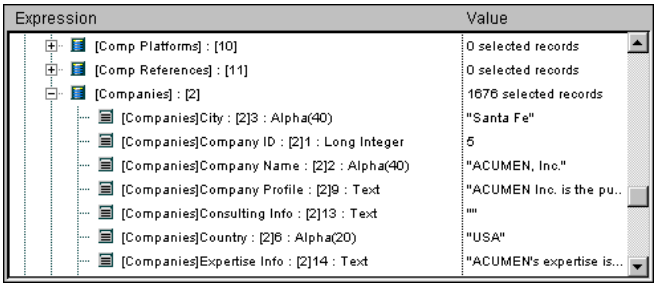
Addtional options are provided by the contextual menu of the Watch pane. To display this menu:
• On Windows, click anywhere in the Watch pane using the **right** mouse button.
• On Macintosh, Control-Click anywhere in the Watch pane.

The contextual 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**.

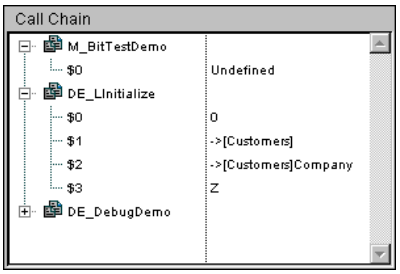
The following is a view of the Watch pane with all options selected:



See Also

Call Chain Pane, Custom Watch Pane, Debugger, Debugger Shortcuts, Source Code 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.

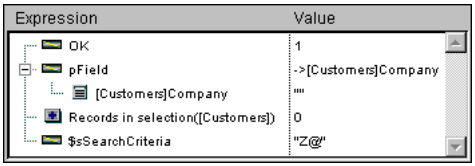
See Also

Custom Watch Pane, Debugger, Debugger Shortcuts, Source Code Pane, 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
- 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.

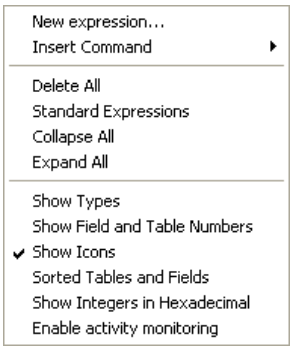
If you no longer need an expression, click on it to select it, then press **Backspace** or **Delete**.

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.

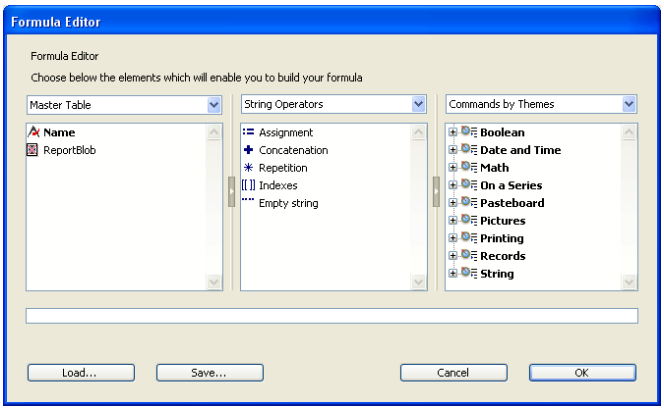
Custom Watch Pane Contextual Menu

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

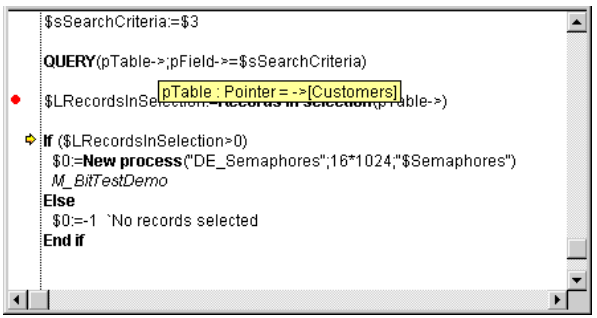
• **Insert Command**: This hierarchical menu item is a shortcut for inserting a command as a new expression, without using the Formula Editor.
• **Delete All**: Deletes all the expressions currently present.
• **Standard Expressions**: Recopies the list of objects in the Expression area.
• **Collapse 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).

See Also

Call Chain Pane, Debugger, Debugger Shortcuts, Source Code Pane, Watch Pane.

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

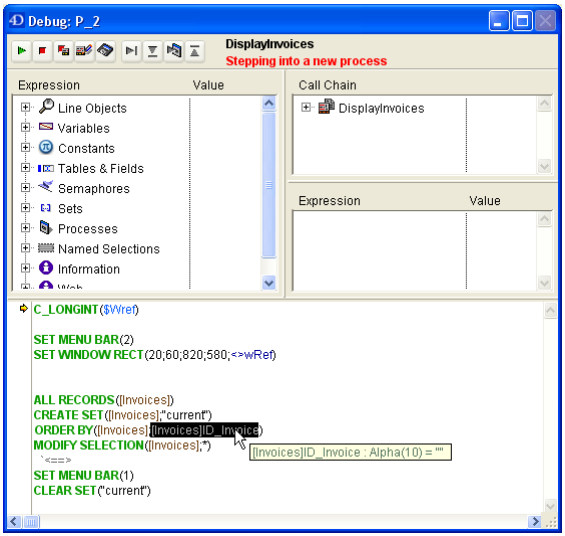
• If the method is too long to fit in the text area, you can scroll to view other parts of the method.
• Moving the mouse pointer over any expression that can be evaluated (field, variable, pointer, array,...) will cause a **Tool Tip** to display the current value of the object or expression and its 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 clicking on the selected text while holding down the **Ctrl** (Windows) or **Command** (Mac OS) key.
• 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. Similarily, 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 TRACE call.

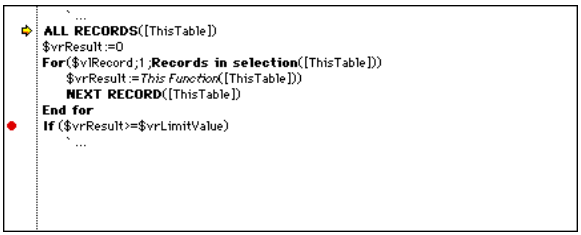
Now, let's say you are executing the following code, with the program counter set to the line ALL RECORDS([ThisTable]):

```
      ` ...
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.

A **red break point** is a **persistent** break point. Once you created it, it "stays." Even though you quit the database, then reopen it later on, the break point will be 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.

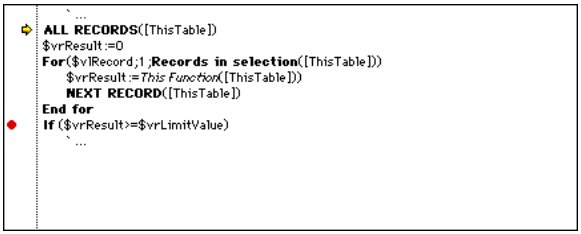
See Also

Break Points, Call Chain Pane, Custom Watch Pane, Debugger, Watch Pane.

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. However, temporary break points can be defined in the debugger only (see below).

In the following figure, a break point has been set, in the debugger, on the line If($vrResult>=$vrLimitValue):



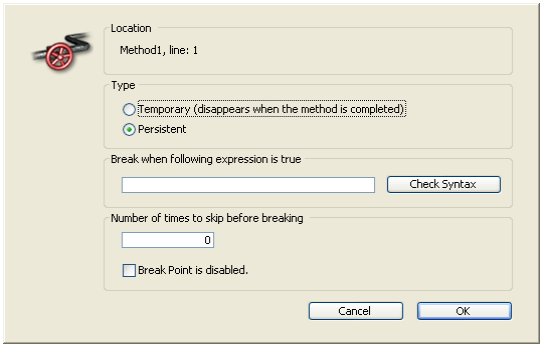
If you click again on the red bullet, the break point is deleted.

Editing a Break Point

Pressing Alt-click (Windows) or Option-click (Macintosh) in the left margin of the Source code pane or of the Method Editor window for a line of code, gives you access to the **Break Point Properties** 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 **Break Point 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.

Type: By default, the debugger lets you create **persistent** break points, depicted by a red bullet in the source code pane of the debugger window. To create a temporary break point, select the **Temporary** option. A temporary break point is useful when you want to break just once in a method. A temporary break point is identified by a green bullet in the source code pane of the Debugger window. You can also set a temporary break point directly in the source code pane by clicking in the left margin while pressing Alt+Shift (Windows) or Option+Shift (Macintosh).

Note: Temporary break points can be set in the debugger only.

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.

Break Point is disabled: If you currently do not need a persistent 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 window.

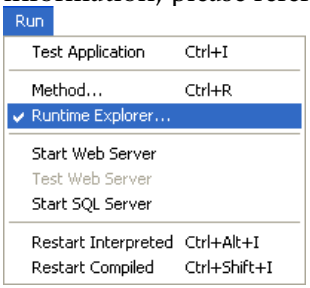
See Also

Break List, Catching Commands, Debugger, Source Code Pane.

The **Break List** is a page of the Runtime Explorer that enables you to manage the persistent Break Points created in the Debugger Window or in the Method Editor.

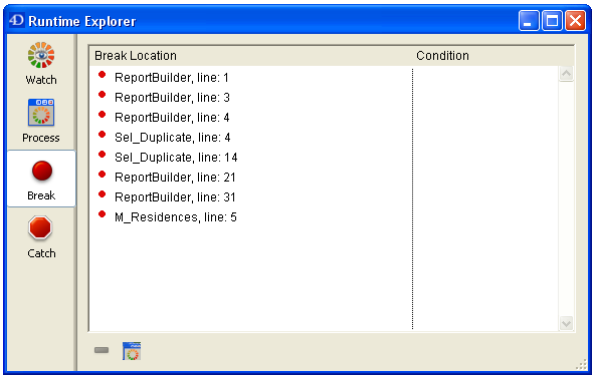
To open the Break List page:

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* manuel.



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 condifition 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 persistent break point from this window. Persistent 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 contextual 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 contextual menu.

Tips

• Adding conditions to break points 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 it automatically skips occurrences that do not match the conditions.

• Disabling a break point 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.

See Also

Break Points, Catching Commands, Debugger, Source Code Pane, Why a Debugger?.

The **Caught Commands List** is a page of the Runtime Explorer that enables you to add additional breaks to your code by catching calls to 4D commands.

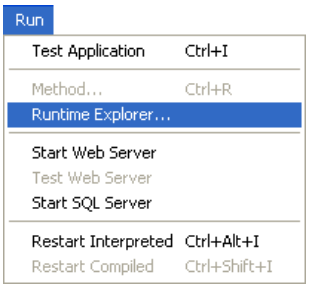
Catching a command enables you to start tracing the execution of any process as soon as a command is called by that process. Unlike a break point, which is located in a particular project method (and therefore triggers a trace exception only when it is reached), the scope of catching a command includes 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.

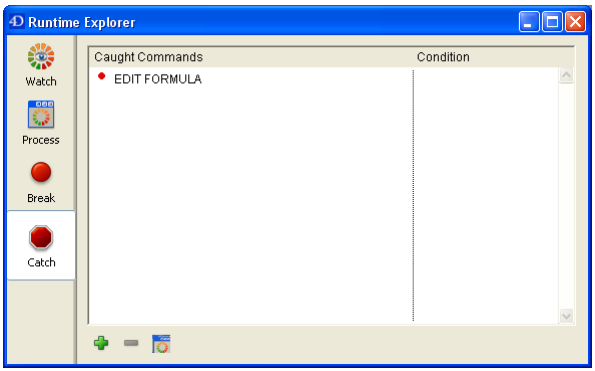
To open the Caught Commands page:

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:



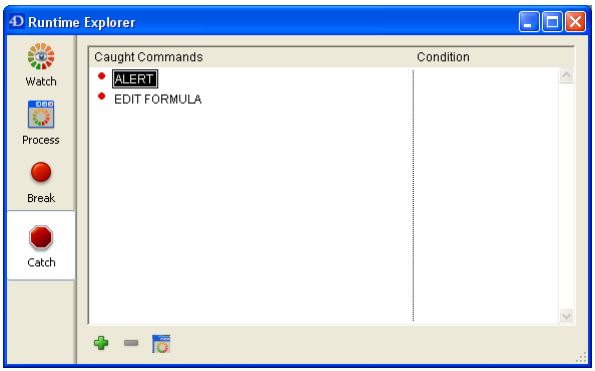
This page lists the commands to be caught during execution. It is composed of two columns:
• The left column displays the Enable/Disable status of the caught command, followed by the name of the command.
• The right column displays the condition associated with the caught command, if any.

Adding a New Command to be Caught

To add a new command:

1. Click on the add button (in the shape of a +) located below the list.
   OR
   Double-click the left mouse button in the Caught Commands list.

In both cases, a new entry is added to the list with the ALERT command as default. The entry is set to the edit mode.



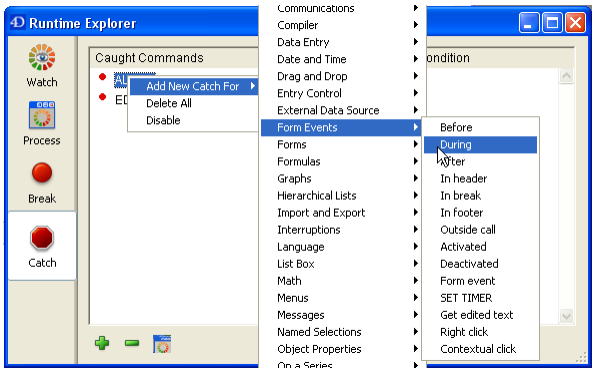
2. Enter the name of the command you want to catch.

3. Press **Enter** or **Return** to validate your choice.

OR

1. Press the right mouse button to display the contextual menu.

2. Select **Add New Catch**, then select the desired command from the command themes and names submenus. A new entry is added with the command you selected.



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. If name you entered does not correspond to an existing 4D command, the entry is set to its previous value. If the entry is a new one, it is reset to ALERT.

Disabling/Enabling a Caught Command

To disable or enable 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. If the entry is in edit mode, press Enter or Return to switch to select mode.

3. Choose **Enable/Disable** from the contextual menu.

Shortcut: Each entry in the list may be disabled/enabled by clicking on the bullet (•). The bullet changes to a dash (–) when disabled.

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. If the entry is in edit mode, press Enter or Return to switch to select mode.

3. Press the **Delete** key or click on the deletion button (in the shape of a '-') located below the list.

Note: To delete all the caught commands, click on the **Delete All** button (third button located below the list) or chosse **Delete All** in the contextual menu.

Setting a Condition for Catching a Command

To set a condition for catching a command:

1. Click on the entry in the right column.

2. Enter a 4D formula (expression, command call or project method) that returns a Boolean value.

Note: To remove a condition, delete its formula.

Tips

• 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 it automatically skips occurrences that do not match the conditions.

• 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.

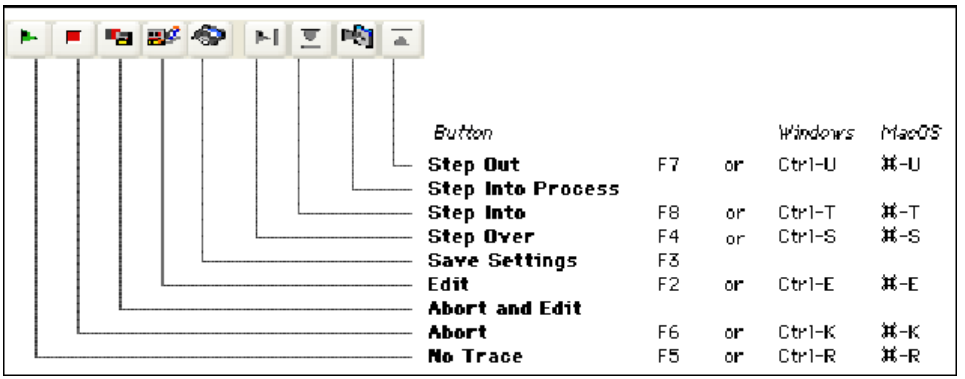
See Also

Break List, Break Points, Debugger.

This section lists all the shortcuts provided by 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 contextual 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 Source Code 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 contextual 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.
→ Click on the selected text while holding down the **Ctrl** (Windows) or **Command** (Mac OS) key copies it to the Custom Watch pane.
→ 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.

See Also

Call Chain Pane, Custom Watch Pane, Debugger, Source Code Pane, Watch Pane.

14

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 or the desktop of the operating system, 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.

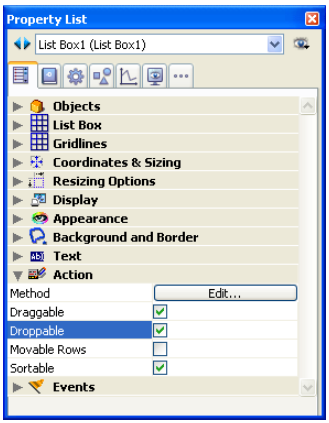
You can also manage dragging and dropping onto the application, outside of any form, using the On Drop Database Method.

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:

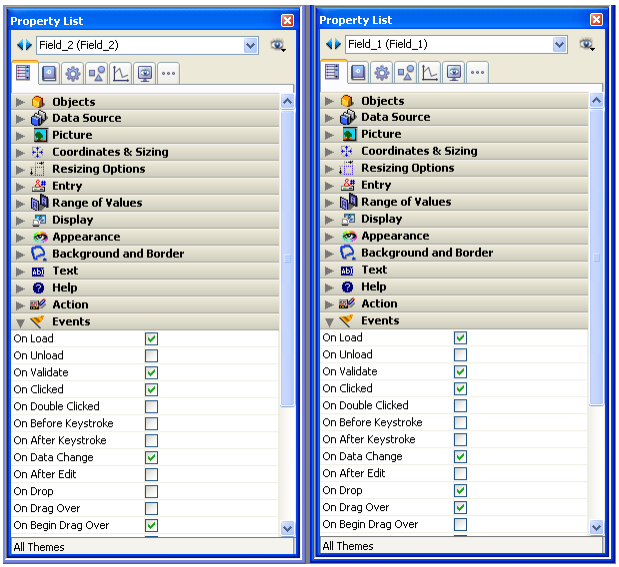


Drag-and-Drop Programmatical Handling

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).
• 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: 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 **source process** (the source object's process), and not in the process of the destination object. This is the only case in which such an execution occurs. The advantages of this type of execution are described at the end of this section.

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.

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.

In this last case, several solutions are available:
• If the On Drag Over event for the destination object method is executed in the context of the source process, you can copy the field data or the record number to an interprocess variable that will be reused during the On Drop event.
• You can get the required data by starting an interprocess communication during the On Drop 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.

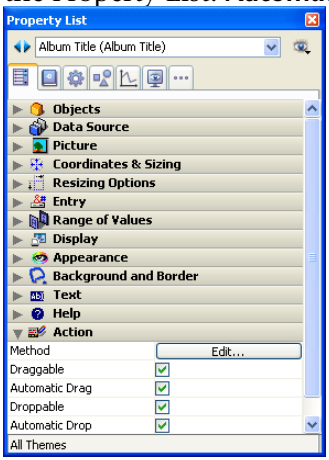
For more information about using the commands of the “Pasteboard” theme for drag and drop operations, please refer to the Managing Pasteboards section.

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 **Command** (Mac OS) key during the action.

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**: When this option is checked, the automatic drag mode is activated for the object. This mode takes priority for pictures, even if the **Draggable** option is checked. In this mode, the On Begin Drag form event is NOT generated.
If you want to "force" the use of the standard drag, hold down the **Alt** (Windows) or **Option** (Mac OS) key during the action.

• **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.
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).

See Also

DRAG AND DROP PROPERTIES, Drop position, Form event, GET PROCESS VARIABLE, Is a list, RESOLVE POINTER, Type.

The On Drop Database Method is available in the 4D single-user and 4D Client 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.

When a drop occurs on the 4D application icon on the destktop, 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
```

See Also

Database Methods.

DRAG AND DROP PROPERTIES (srcObject; srcElement; srcProcess)

Parameter	Type		Description
srcObject	Pointer	←	Pointer to drag-and-drop source object
srcElement	Number	←	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	Number	←	Source process number

Description

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. If the source object is a field, you must get the value from the source process via interprocess communication or handle that particular case while responding to the On Drag Over event (see below). However, 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.

Tip: During an On Drag Over event, the object method of the destination object is executed within the context of the source object's process. If the source object of an interprocess drag and drop is a field, you can use the opportunity of this event to copy the source data into an interprocess variable. By doing so, later on, during the On Drop event, you will not have to initiate an interprocess communication with the source process in order to get the value of the field that was dragged. If an interprocess drag and drop involves a variable as source object, you can use the GET PROCESS VARIABLE command during the On Drop event.

Examples

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
                ` If the destination element was beyond the dragged element
            DELETE FROM ARRAY($1->;$vlSrcElem)
            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
```

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 Alpha Field)
                  ` Alphanumeric Field is OK
               $0:=0   ` Copy the value now into an IP variable
               <>vtDraggedData:=$vpSrcObj->
            : ($vlSrcType=Is Text)
                  ` Text Field or Variable is OK
               $0:=0
               RESOLVE POINTER($vpSrcObj;$vsVarName;$vlTableNum;
                                                                  $vlFieldNum)
                  ` If it is a field
               If (($vlTableNum>0) & ($vlFieldNum>0))
                     ` Copy the value now into an IP variable
                  <>vtDraggedData:=$vpSrcObj->
               End if
```

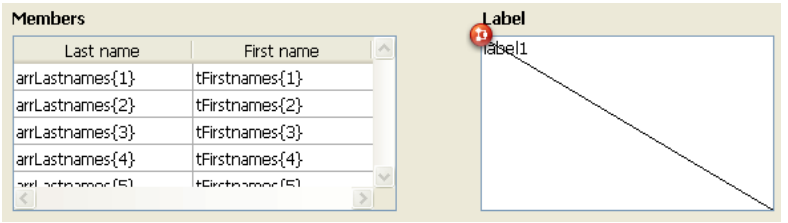
```
            : ($vlSrcType=Is String Var)
                  ` String Variable is OK
               $0:=0
            : (($vlSrcType=String array) | ($vlSrcType=Text array))
                  ` String and Text Arrays are OK
               $0:=0
            : (($vlSrcType=Is LongInt) | ($vlSrcType=Is Real)
               If (Is a list($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)
      RESOLVE POINTER($vpSrcObj;$vsVarName;$vlTableNum;$vlFieldNum)
         ` If it is field
      If (($vlTableNum>0) & ($vlFieldNum>0))
            ` Just grab the IP variable set during the On Drag Over event
         $vtDraggedData:=<>vtDraggedData
      Else
            ` 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
        End if
            ` If there is actually something to drop (the source object may be empty)
        If ($vtDraggedData # "")
                ` Check that the length of the text variable will not
                ` exceed 32,000 characters
            If ((Length($1->)+Length($vtDraggedData))<=32000)
                $1->:=$1->+$vtDraggedData
            Else
                BEEP
                ALERT("The drag and drop cannot be completed because the text would
                                                                become too long.")
            End if
        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
```

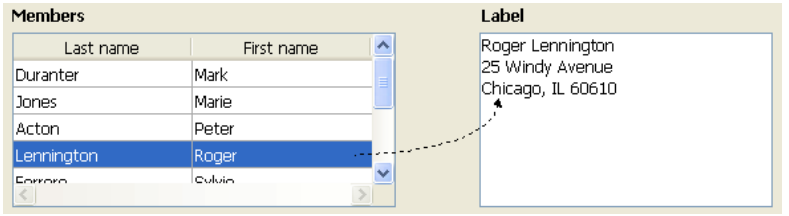
3. We want to fill a text area (for example, a label) with data dragged from a list box.



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:



See Also

Drag and Drop, Drop position, Form event, GET PROCESS VARIABLE, Is a list, RESOLVE POINTER.

Drop position

Drop position {(columnNumber)} → Number

Parameter	Type		Description
columnNumber	Longint	←	List box column number or -1 if the drop occurs beyond the last column
Function result	Number	←	• Number (array/list box) or • Position (hierarchical list) or • Position in string (text/combo box) of destination item or -1 if drop occurred beyond the last array element or list item

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 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 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 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 or a text, the command returns -1.

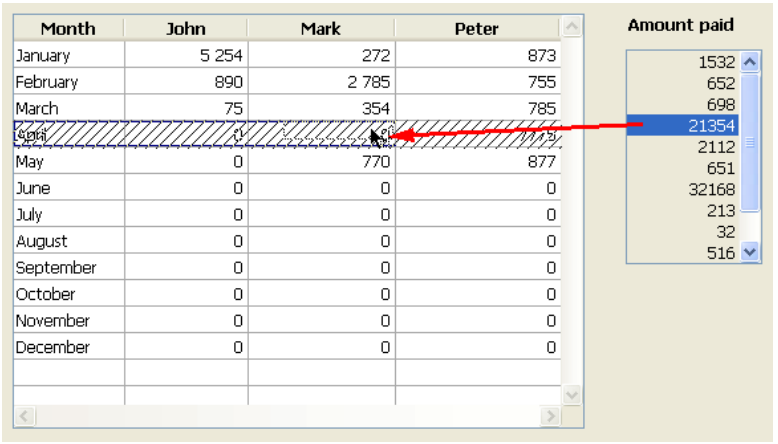
Important: A form object accepts dropped data if its **Droppable** property has been selected.

Also, its object method must be activated for On Drag Over and/or On Drop, in order to process these events.

Examples

1. See the examples for the DRAG AND DROP PROPERTIES command.

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:



Month	John	Mark	Peter
January	5 254	272	873
February	890	2 785	755
March	75	354	785
April	0	0	0
May	0	770	877
June	0	0	0
July	0	0	0
August	0	0	0
September	0	0	0
October	0	0	0
November	0	0	0
December	0	0	0

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
```

See Also

Drag and Drop, DRAG AND DROP PROPERTIES.

15

Entry Control

ACCEPT

Parameter	Type	Description
This command 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, ADD SUBRECORD, or 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.

See Also

CANCEL.

CANCEL

Parameter	Type	Description

This command 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, ADD SUBRECORD, or 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.

See Also

ACCEPT, PAGE BREAK, Print form.

System Variables and Sets

When the CANCEL command is executed (form or printing cancelled), the system variable *OK* is set to 0.

EDIT ITEM ({*; }object{; item})

Parameter	Type		Description
*	*	→	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	Number	→	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 MODIFY SELECTION or DISPLAY SELECTION commands.

If the command is used with an enterable object that is not a list, it then acts the same as the GOTO AREA 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 hierarchical lists, the EDIT ITEM command automatically causes the list to be redrawn. As a result, you should not call the REDRAW LIST command when the EDIT ITEM command is used.

• In subforms and list forms, the command changes the first field of a specified row to edit mode, in the order of entry.

Examples

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")
```

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:

```
$vlRowNum:=Size of array(Array1)+1
INSERT LISTBOX ROW(*;"MyListBox";$vlRowNum)
Array1{$vlRowNum}:="New value 1"
Array2{$vlRowNum}:="New value 2"
EDIT ITEM(Array2;$vlRowNum)
```

3. The following example allows changing the first field of the last subrecord in the subselection to editing mode:

```
LAST SUBRECORD([Children])
EDIT ITEM(*;"Subform")
```

See Also

GOTO AREA, INSERT IN LIST, SET LIST ITEM.

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 command Keystroke.

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 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 type-ahead 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.

Examples

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.

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.

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
```

See Also

Form event, Get edited text, Keystroke.

GOTO AREA ({*; }object)

Parameter	Type		Description
*	*	→	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 AREA 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).

Note: This command only functions in input forms. It has no effect on data entry areas located in subform List forms.

Examples

1. The GOTO AREA command can be used in both ways:

```
GOTO AREA ([People]Name)  ` Field Reference
GOTO AREA (*;"AgeArea")  ` Object Name
```

2. You don't want any object of the form to have the focus:

```
GOTO AREA (*;"")
```

3. See the example for the command REJECT.

See Also

REJECT.

Keystroke → String

Parameter	Type		Description
This command does not require any parameters			
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 event form. 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 type-ahead areas

Examples

1. See examples for the command FILTER KEYSTROKE.

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
```

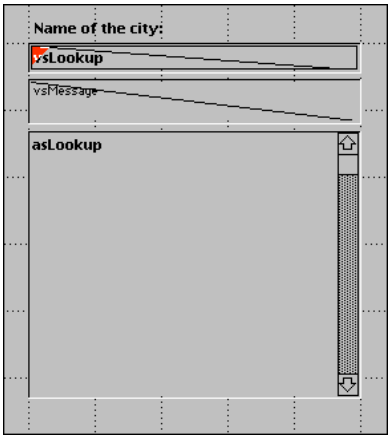
```
                ` The text selection is the text cursor
            Case of
                    ` The text cursor is currently at the begining of the text
                : ($vlStart<=1)
                        ` Insert the character at the begining of the text
                    $vtNewValue:=Keystroke+$vtNewValue
                        ` 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:



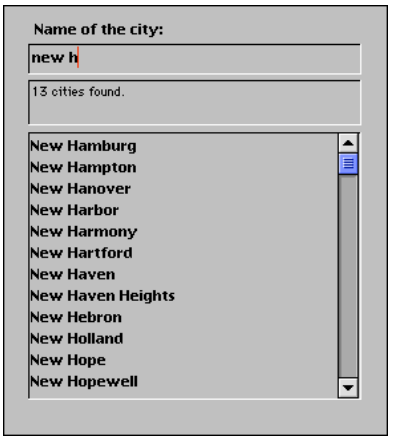
It is composed of the following objects: an enterable area vsLookup, a non-enterable area vsMessage, and a scrollable area asLookup. While entering characters in vsLookup, the method for that object performs a query on a [US Zip Codes] table, allowing the user to find US cities by typing only the first 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.

See Also

FILTER KEYSTROKE, Form event, Get edited text.

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 the 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. The cursor stays in the field area. This form of REJECT forces the user to enter a correct value. It must be used immediately following a modification to the field. You can test for modification by using the Modified function. You can also use REJECT in the object method for the data entry area. This command has no effect on fields in subform areas.

You must put either form of the REJECT command in the form method or object method for the form that is being modified. 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.

You can use HIGHLIGHT TEXT to select the data in the field that is being rejected.

Examples

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
        ` If it is a check with no number...
    : (([Operation]Transaction="Check") & ([Operation]Check Number = ""))
        ALERT ("Please fill in the check number.") ` Alert the user
        REJECT ` Reject the entry
        GOTO AREA ([Operation]Check Number) ` Go to the check number field
End case
```

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:

```
If ([Employees]Salary<10000)
    ALERT ("Salary must be greater than $10,000")
    REJECT ([Employees]Salary)
End if
```

See Also

ACCEPT, CANCEL, GOTO AREA.

16

External Data Source

Overview

This theme contains commands that allow 4D to access, via standard protocols, data stored in other applications. In the current version of 4D, only ODBC commands are available.

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, etc.) via SQL language.

Note: 4D also allows data to be imported from and exported to an ODBC source in Design mode. For more information, please refer to the 4D Design Reference manual.

The high-level ODBC commands in the "External Data Source" theme 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**.

How built-in ODBC commands work

The built-in ODBC commands of 4D implement the following principles:

• The scope of a connection is the process. If you want to manage several simultaneous connections, you must start a process by ODBC LOGIN.
The ODBC CANCEL LOAD command can be used to execute several SELECT requests in the same connection.

• You can use most of the commands of this theme with the 4D internal SQL kernel.

• You can intercept any ODBC errors generated during the execution of one of the ODBC commands using the ON ERR CALL command. The ODBC GET LAST ERROR command can be used in this case to obtain additional information.

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

Referencing 4D expressions in ODBC requests
4D provides two ways for inserting 4D expressions (variables, arrays, fields, pointers, valid expressions) into ODBC requests: direct association and the setting of parameters using ODBC 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 ":".
Examples:

```
ODBC EXECUTE("INSERT INTO emp (empnum,ename) VALUES
(<<vEmpnum>>,<<vEname>>)")
ODBC EXECUTE("SELECT age FROM People WHERE name= :vName")
```

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 ODBC 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 ODBC SET PARAMETER command.

1. This example executes an ODBC request 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
ODBC LOGIN("mysql";"root";"")
SQLStmt:="insert into app_testTable (alpha_field, longint_field) VALUES
                                        (<<MyTextArray>>, <<MyLongintArray>>)"
ODBC EXECUTE(SQLStmt)
```

2. This example can be used to execute an ODBC request that directly uses the associated 4D fields:

```
ALL RECORDS([Table 2])
ODBC LOGIN("mysql";"root";"")
SQLStmt:="insert into app_testTable (alpha_field, longint_field) VALUES
                                        (<<[Table 2]Field1>"+">,<<[Table 2]Field2>>)"
ODBC EXECUTE(SQLStmt)
```

3. This example lets you execute an ODBC query by directly passing a variable via a dereferenced pointer:

```
C_LONGINT($vLong)
C_POINTER($vPointer)
$vLong:=1
$vPointer:=->$vLong
ODBC LOGIN("mysql";"root";"")
SQLStmt:="SELECT Col1 FROM TEST WHERE Col1=:$vPointer"
ODBC EXECUTE(SQLStmt)
```

Retrieving values in 4D

Retrieving values in 4D that result from ODBC requests is carried out in two ways:

- Using the additional parameters of the ODBC EXECUTE command (recommended solution).
- Using the INTO clause in the SQL query itself (solution reserved for special cases).

ODBC CANCEL LOAD

Parameter	Type	Description
This command does not require any parameters		

Description

The ODBC 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 ODBC LOGIN command.

Example

In this example, two requests are executed in the same connection:

```
C_BLOB(Myblob)
C_TEXT(MyText)
ODBC LOGIN("mysql";"root";"")

SQLStmt:="SELECT blob_field FROM app_testTable"
ODBC EXECUTE(SQLStmt;Myblob)
While(Not(ODBC End selection))
   ODBC LOAD RECORD
End while

   `Resetting of cursor
ODBC CANCEL LOAD

SQLStmt:="SELECT Name FROM Employee"
ODBC EXECUTE(SQLStmt;MyText)
While(Not(ODBC End selection))
   ODBC LOAD RECORD
End while
```

See Also

ODBC LOAD RECORD, ODBC LOGIN.

System Variables or Sets

If the command has been correctly executed, the system variable OK returns 1. Otherwise, it returns 0.

ODBC End selection → Boolean

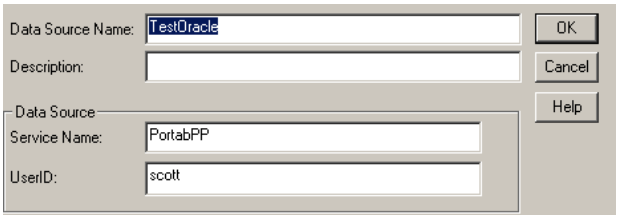
Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	Result set boundaries reached

Description

The ODBC 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)

ODBC LOGIN("TestOracle";"scott";"tiger")
If (OK=1)
    ODBC EXECUTE("SELECT ename FROM emp";vName)
    While(Not(ODBC End selection))
        ODBC LOAD RECORD
    End while
    ODBC LOGOUT
End if
```

This code will return in the 4D vName variable the emp names (*ename*) stored in the table named *emp*.

ODBC EXECUTE (sqlStatement{; boundObj}{; boundObj2; ...; boundObjN})

Parameter	Type		Description
sqlStatement	Text	→	SQL command to execute
boundObj	Variable | Field	←	Receives result (if necessary)

Description

The ODBC EXECUTE command is used to execute an SQL command and to bind the result to 4D objects (arrays, variables or fields).

A valid connection is required 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.

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, please refer to the External Data Source Commands section.

Examples

1. In this example, we will get the ename column of the emp table of the external data source. The result is stored in the [Employee]Name 4D field. 4D records will be created automatically:

```
SQLStmt:="SELECT ename FROM emp"
ODBC EXECUTE(SQLStmt;[Employee]Name)
ODBC LOAD RECORD(ODBC All Records)
```

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:

```
ODBC LOGIN("mysql";"root";"")
SQLStmt:="SELECT alpha_field FROM app_testTable"
START TRANSACTION
ODBC EXECUTE(SQLStmt;[Table 2]Field1)
While(Not(ODBC End Selection))
    ODBC LOAD RECORD
    ...  `Place the data validation code here
End while
VALIDATE TRANSACTION  `Validation of the transaction
```

3. In this example, we want to get the ename column of the emp table of the external 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"
ODBC EXECUTE(SQLStmt;aName)
While(Not(ODBC End Selection))
    ODBC LOAD RECORD(10)
End while
```

4. In this example, we want to get the ename and job of the emp table for a specific ID (WHERE clause) of the external 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"
ODBC EXECUTE(SQLStmt;vName;vJob)
ODBC LOAD RECORD
```

See Also

ODBC LOAD RECORD.

System Variables or Sets

If the command has been executed correctly, the system variable OK returns 1. Otherwise, it returns 0.

ODBC EXPORT (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)
*	*	→	Display of export dialog box and project update

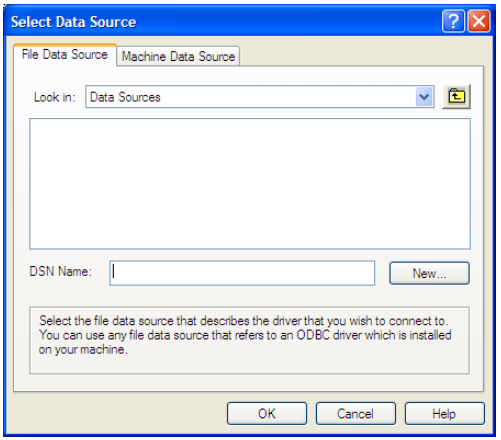
Description

The ODBC EXPORT command is used to export data in the sourceTable table of an external ODBC source. The connection parameters (source name, user and password) are included in the project BLOB.

Notes:
• The project contains all the export parameters, in particular the data source and the exported tables and fields. You set these parameters in the ODBC export dialog box, then you can save them in a file on disk if necessary. For more information, refer to the *Design Reference* manual.
• The projects generated in the ODBC export dialog box are not compatible with the commands or the standard export dialog box of 4D.
• 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, ODBC EXPORT displays a dialog box for selecting the ODBC data source:

Windows



Mac OS

Once you have selected the source, the 4D ODBC export dialog box appears, allowing the user to configure the operation. If the user clicks **Cancel** in either of the two dialog boxes, execution is stopped and the system variable OK is set to 0.

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. You can also use the ODBC EXPORT 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.

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.

See Also

ODBC IMPORT.

System Variables or 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.

ODBC 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 ODBC 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.

See Also

ON ERR CALL.

ODBC GET OPTION (option; value)

Parameter	Type		Description
option	Longint	→	Option number
value	Longint	←	Option value

Description

The ODBC 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 ODBC SET OPTION command.

See Also

ODBC SET OPTION.

System Variables or Sets

If the command was properly executed, the system variable OK is set to 1. Otherwise, it is set to 0.

ODBC IMPORT (sourceTable{; project{; *}})

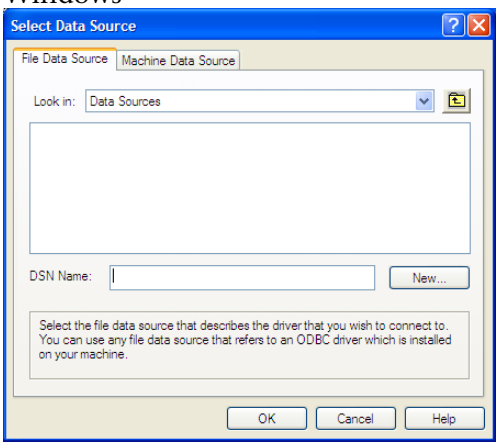
Parameter	Type		Description
sourceTable	String	→	Name of table in ODBC data source
project	BLOB	→	Contents of import project
		←	New contents of import project (if * is passed)
*	*	→	Display of import dialog box and project update

Description

The ODBC IMPORT command is used to import data from the sourceTable table of an external ODBC source. The connection parameters (source name, user and password) are included in the project BLOB.

Notes:
• The project contains all the import parameters, in particular the data source and target tables and fields. You set these parameters in the ODBC import dialog box, then you can save them in a file on disk if necessary. For more information, refer to the *Design Reference* manual.
• The projects generated in the ODBC import dialog box are not compatible with the commands or the standard import dialog box of 4D.
• 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, ODBC IMPORT displays a dialog box for selecting the ODBC data source:
Windows



Mac OS

Once you have selected the source, the 4D ODBC import dialog box appears, allowing the user to configure the operation. If the user clicks **Cancel** in either of the two dialog boxes, execution is stopped and the system variable OK is set to 0.

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. You can also use the ODBC IMPORT 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.

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.

See Also

ODBC EXPORT.

System Variables or 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.

ODBC LOAD RECORD {(numRecords)}

Parameter	Type		Description
numRecords	Integer	→	Number of records to load

Description

The ODBC LOAD RECORD command retrieves one or more record(s) in 4D coming from the ODBC source open in the current connection.

The optional numRecords parameter is used to set the number of records to retrieve:
• If you omit this parameter, the command will retrieve 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 will retrieve numRecords records.
• If you pass the ODBC All Records constant (value -1), the command will retrieve all the records of the table.

Note: These last two parameters are only meaningful if the data retrieved are associated with arrays or with 4D fields.

See Also

ODBC CANCEL LOAD, ODBC EXECUTE.

System Variables or Sets

If the command has been executed correctly, the system variable OK returns 1. Otherwise, it returns 0.

ODBC LOGIN{(dataEntry; userName; password)}

Parameter	Type		Description
dataEntry	String	→	Name of the data source entry in the ODBC Manager
userName	String	→	Name of the user registered in the data source
password	String	→	Password of the user registered in the data source

Description

The ODBC LOGIN command allows you to connect to an external ODBC data source or to the 4D internal SQL kernel.

Note: The ODBC (Open DataBaseConnectivity) standard defines a library of standard functions. These functions allow an application like 4D to accesss any ODBC-compatible data source (databases, spreadsheets, etc.) using SQL.

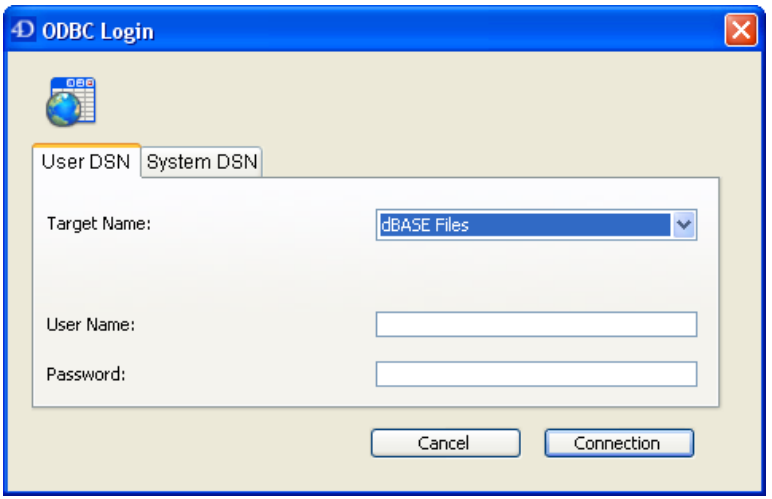
The dataEntry parameter contains the name of the data source as entered in the ODBC driver manager.
To open a connection with the 4D internal SQL kernel, pass the SQL_INTERNAL constant in the dataEntry parameter.

Note: It is not necessary to open a connection using this command if you intend to access the 4D SQL environment using the Begin SQL/End SQL keywords or the QUERY BY SQL command. The connection must be initialized only for the use of other 4D ODBC commands.

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”.

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.

Examples

1. This statement will bring up the ODBC Manager dialog box:

```
ODBC LOGIN
```

2. This statement will connect to the ODBC data source named “MyOracle” using Scott/tiger as the name/password :

```
ODBC LOGIN("MyOracle";"Scott";"tiger")
```

3. Open a connection with the 4D internal SQL kernel:

```
ODBC LOGIN(SQL_INTERNAL;$user;$password)
```

See Also

ODBC LOGOUT.

System Variables or Sets

If the connection is successful, the system variable OK is set to 1; otherwise, it is set to 0.

ODBC LOGOUT

Parameter	Type	Description
This command does not require any parameters		

Description

The ODBC LOGOUT command closes the connection of the current process (if applicable). If there is no connection, the command does nothing.

See Also

ODBC LOGIN.

System Variables or 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.

ODBC SET OPTION (option; value)

Parameter	Type		Description
option	Longint	→	Number of option to set
value	Longint	→	New value of option

Description

The ODBC SET OPTION command is used to modify the value of the option passed in option.

option can have one of the following values, located in the "External Data Source" theme:

Constant	Description and possible values
OBDC Asynchronous (1)	0 = Synchronous connection (default value), 1 (or value other than 0) = Asynchronous connection
ODBC Max Rows (2)	Maximum number of rows in resulting group (used for previews)
ODBC Max Data Length (3)	Maximum length of data returned
ODBC Query Time Out (4)	Maximum timeout awaiting response when executing the ODBC EXECUTE command. Values: time in seconds By default: no timeout
ODBC Connection Time Out (5)	Maximum timeout awaiting response when executing the ODBC LOGIN command. This value must be set before opening the connection in order to be taken into account Possible values: time in seconds By default: no timeout

Note: When you work with the internal SQL kernel of 4D, the ODBC Asynchronous option serves no purpose due to the fact that this type of connection is always synchronous.

See Also

ODBC GET OPTION.

System Variables or Sets

If the command was properly executed, the system variable OK returns 1. Otherwise, it returns 0.

ODBC 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 ODBC SET PARAMETER command allows the use of a 4D variable, array or field value in ODBC 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 External Data Source 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 "External Data Source" theme:

Constant	Type	Value
OBDC Param In	Longint	1
OBDC Param In Out	Longint	2
OBDC Param Out	Longint	4

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 ODBC 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.

Examples

1. This example is used to execute an ODBC request which calls the associated 4D variables directly:

```
C_TEXT(MyText)
C_LONGINT(MyLongint)

ODBC 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
    ODBC EXECUTE(SQLStmt)
End for
```

2. Same example as the previous one, but using the ODBC SET PARAMETER command:

```
C_TEXT(MyText)
C_LONGINT(MyLongint)

ODBC LOGIN("mysql";"root";"")
SQLStmt:="insert into app_testTable (alpha_field, longint_field) VALUES (?,?)"
For (vCounter;1;10)
    MyText:="Text"+String(vCounter)
    MyLongint:=vCounter
    ODBC SET PARAMETER(MyText;ODBC Param In)
    ODBC SET PARAMETER(MyLongint;ODBC Param In)
    ODBC EXECUTE(SQLStmt)
End for
```

3. This example is used to execute an ODBC request which uses the associated 4D arrays directly:

```
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
ODBC LOGIN("mysql";"root";"")
SQLStmt:="insert into app_testTable (alpha_field, longint_field) VALUES
                                        (<<MyTextArray>>, <<MyLongintArray>>)"
ODBC EXECUTE(SQLStmt)
```

4. This example is used to execute an ODBC request which uses the associated 4D fields directly:

```
ALL RECORDS([Table 2])
ODBC LOGIN("mysql";"root";"")
SQLStmt:="insert into app_testTable (alpha_field, longint_field) VALUES
                                    (<<[Table 2]Field1>"+">, <<[Table 2]Field2>>)"
ODBC EXECUTE(SQLStmt)
```

System Variables or Sets

If the command has been executed correctly, the system variable OK returns 1. Otherwise, it returns 0.

17

Form Events

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Activate event.

Activated → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	Returns TRUE if the execution cycle is an activation

Description

The Activated command returns TRUE in a form method when the window containing the form becomes the frontmost window of the frontmost process.

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. This is done automatically when a database is converted.

See Also

Deactivated, Form event.

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Validate event.

After → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

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.

See Also

Form event.

version 3

Compatibility Note

This command has been kept in 4D for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Load event.

Before → Boolean

Parameter	Type	Description

This command does not require any parameters

Description

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.

See Also

Form event.

Contextual click → Boolean

Parameter	Type		Description
This command does not require any parameters			
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 contextual menu:

```
If(Contextual click)
    If (Pop up menu("True;False")=1)
        myArray{myArray}:="True"
    Else
        myArray{myArray}:="False"
    End if
End if
```

See Also

Form event, Right click.

Deactivated

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Deactivate event.

Deactivated → Boolean

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

Activated, Form event.

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an event such as On Clicked.

During → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

In order for the During execution cycle to be generated, make sure that the appropriate event properties, such as On Clicked, for the form and/or the objects have been selected in the Design environment.

See Also

Form event.

Form event → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	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.

Generic events
The following events can be generated for any form or object:

Constant	Value	Description
On Load	1	The form is about to be displayed or printed
On Unload	24	The form is about to be exited and released
On Validate	3	The record data entry has been validated
On Clicked	4	A click occurred on an object
On Double Clicked	13	A double click occurred on an object
On Before Keystroke	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 After Keystroke	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 Edit	45	The contents of the enterable object that has the focus has just been modified
On Getting Focus	15	A form object is getting the focus
On Losing Focus	14	A form object is losing the focus
On Activate	11	The form’s window becomes the frontmost window
On Deactivate	12	The form’s window ceases to be the frontmost window

On Outside Call	10	The form received a CALL PROCESS call
On Drop	16	Data has been dropped onto an object
On Drag Over	21	Data could be dropped onto an object
On Begin Drag Over	46	An object is being dragged
On Mouse Enter	35	The mouse cursor enters the graphic area of an object
On Mouse Move	37	The mouse cursor moves (at least one pixel) within the graphic area of an object
On Mouse Leave	36	The mouse cursor leaves the graphic area of an object
On Menu Selected	18	A menu item has been chosen
On Data Change	20	Object data has been modified
On Plug in Area	19	An external object requested its object method to be executed
On Header	5	The form's header area is about to be printed or displayed
On Printing Detail	23	The form's detail area is about to be printed
On Printing Break	6	One of the form's break areas is about to be printed
On Printing Footer	7	The form's footer area is about to be printed
On Close Box	22	The window's close box has been clicked
On Display Detail	8	A record is about to be displayed in a list
On Open Detail	25	A record is double clicked and you are going to the input form
On Close Detail	26	You left the input form and are going back to the output form
On Selection Change	31	• List box: the current selection of rows or columns is modified • Records in list: the current record or the current selection of rows is modified in a list form or subform • Hierarchical list: the selection in the list is modified following a click or a keystroke
On Load Record	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 Timer	27	The number of ticks defined by the SET TIMER command has passed
On Resize	29	The form window is resized

List box
The following events are only generated for List boxes:

Constant	Value	Description
On Before Data Entry	41	A list box cell is about to change to editing mode
On Column Moved	32	A list box column is moved by the user via drag and drop
On Row Moved	34	A list box row is moved by the user via drag and drop
On Column Resize	33	The width of a list box column is modified
On Header Click	42	A click occurs in a column header of the list box
On After Sort	30	A standard sort has just been carried out in a list box column

3D buttons
The following events are only generated for 3D buttons:

Constant	Value	Description
On Long Click	39	A 3D button is clicked and the mouse button remains pushed for a certain lapse of time
On Arrow Click	38	The “arrow” area of a 3D button is clicked

Hierarchical lists
The following events are only generated for hierarchical lists:

Constant	Value	Description
On Expand	43	An element of the hierarchical list has been expanded using a click or a keystroke
On Collapse	44	An element of the hierarchical list has been collapsed using a click or a keystroke

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 or On 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 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.

• Records in list: The sequence of calls to methods and form events in the list forms displayed via MODIFY SELECTION / DISPLAY 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.

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) (*)	Yes	Involved object only
On Double Clicked	Yes (if clickable) (*)	Yes	Involved object only
On Before Keystroke	Yes (if keyboard enterable) (*)	Yes	Involved object only
On After Keystroke	Yes (if keyboard 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
Sur début glisser	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 Menu Selected	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	Never	Yes	None
On Close Detail	Never	Yes	None
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 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 After Sort	Yes (List box)	Never	Involved object only
On Long Click	Yes (3D button)	Yes	Involved object only
On Arrow Click	Yes (3D button)	Yes	Involved object only
On Expand	Yes (Hier. list)	Never	Involved object only
On Collapse	Yes (Hier. list)	Never	Involved object only

(*) For more infomation, see the "Events, Objects and Properties" section below.
(**) Refer to the "Drag and Drop" section 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
- 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.
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.

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.

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 and On After Keystroke 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
• List boxes.

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: Beginning with version 2004.2 of 4D, 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 andOn 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;
- Dragging a value (action similar to copy);
- 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.

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 subtables and 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.

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.

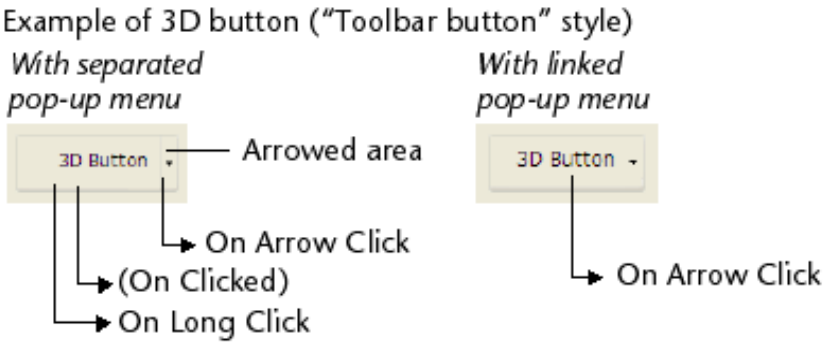
3D buttons

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 Arrow 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 additional actions in relation to the primary button action.
4D allows you to manage this type of button using the On Arrow 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

Seven 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 MOVED LISTBOX 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 (using the mouse or by programming using the SET LISTBOX COLUMN WIDTH command).

• 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.
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 After Sort: This event is generated just after a standard sort is performed (however, 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 column that was sorted.

Hierarchical lists
In addition to generic events, three 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.

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

Examples

In all the examples discussed here, it is assumed that the event properties of the forms and objects have been selected appropriately.

1. This example sorts a selection of subrecords for the subtable [Parents]Children before a form for the [Parents] table is displayed on the screen:

```
    ` Method of a form for the [Parents] table
Case of
    : (Form event=On Load)
        ORDER SUBRECORDS BY([Parents]Children;[Parents]Children'First name;>)
        ` ...
End case
```

2. 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
```

3. 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 STRING(31;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
```

4. 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.
                ` Note that the mouse button is still pressed, the only effect while
                ` accepting the drag is to let 4D highlight the object so the user
                ` knows the source data could be dropped onto it
            $0:=0
        Else
```

```
            ` If so, refuse the drag
        $0:=-1
            ` In this case, the object is not highlighted
    End if
: (Form event=On Drop)
        ` The source data has been dropped on the object, we therefore need to copy it
        ` into the object
        ` Get the information about the source object
    DRAG AND DROP PROPERTIES ($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.

5. 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
```

6. 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)
```

7. 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.

8. 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 STRING (...;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
```

9. 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 STRING (...;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
```

10. 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
```

11. 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
```

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
```

See Also

CALL PROCESS, Current form table, DRAG AND DROP PROPERTIES, FILTER KEYSTROKE, Get edited text, Keystroke, SET TIMER.

Get edited text → Text

Parameter	Type		Description
This command does not require any parameters			
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.

Note: To be in accordance with the new form event On After Keystroke (introduced in version 6.5 of 4D), the existing event On Keystroke has been renamed, and is now called On Before Keystroke.

When used in a context other than text entry in a form object, this function returns an empty string.

Examples

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
```

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.).

See Also

Form event.

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Printing Break event.

In break → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

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.

See Also

During, In footer, In header.

In footer

version 3

Compatibility Note

This command has been kept for compatibility reason. Starting with version 6, you may want to start using the command Form event and check if it returns an On Printing Footer event.

In footer → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

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.

See Also

During, In break, In header.

In header

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Header event.

In header → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

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.

See Also

During, In break, In footer.

Outside call

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Outside call event.

Outside call → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

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.

See Also

CALL PROCESS, Form event.

Right click

Right click → Boolean

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

Contextual click, Form event.

SET TIMER (tickCount)

Parameter	Type		Description
tickCount	Longint	→	Tickcount

Description

The SET TIMER command allows you to activate the On Timer form event and to set, for the current 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.
4D's Web server can take advantage of this command as well as the On Timer form event to resend 4D forms. This feature allows you to obtain HTML pages updated in "real time" while saving bandwidth. Actually, updating a form in this case is not automatic; you must call the REDRAW command. You can then optimize the system by calling REDRAW only when the data has been modified.
Only browsers that interpret JavaScript allow you to automatically redraw pages. The laps defined by SET TIMER will be used by the browser and by the timeout of the Web process. The laps must be a few seconds (5 being a practical value). For more information, please refer to the second example shown below.

To procedurally disable the triggering of the On Timer form event, call SET TIMER again and pass 0 in tickCount.

Examples

1. 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
```

2. Let us imagine that you want your Web server to update a 4D form displayed on the Web browser every five seconds. To do so, write the following form method:

```
If (Form event=On Load)
   SET TIMER(60*5)
End if

If (Form event=On Timer)
   ... `You can place a test here to see if the data is being modified and to
      `execute the following line only if this is true.
   REDRAW ([MyTable])
End if
```

See Also

Form event, REDRAW.

18

Forms

Current form page

Current form page → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Number of currently displayed form page

Description

The Current form page command returns the number of the currently displayed form page.

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=...)
                  : (Current form page=1)
                     ` Do appropriate action for page 1
                  : (Current form page=2)
                     ` Do appropriate action for page 2
                     ` ...
```

```
                        : ($vlItemNumber=...)
                            ` ...
                    End case
                : ($vlMenuNumber=...)
                    ` ...
            End case
        : (Form event=On Outside call)
            Case of
                : (Current form page=1)
                    ` Do appropriate reply for page 1
                : (Current form page=2)
                    ` Do appropriate reply for page 2
            End case
            ` ...
End case
```

See Also

FIRST PAGE, GOTO PAGE, LAST PAGE, NEXT PAGE, PREVIOUS PAGE.

FIRST PAGE

Parameter	Type	Description

This command does not require any parameters

Description

FIRST PAGE changes the currently displayed form page to the first form page. If a form is not being displayed, or if the first form page is already displayed, FIRST PAGE does nothing.

Example

The following example is a one-line method called from a menu command. It displays the first form page:

```
FIRST PAGE
```

See Also

Current form page, GOTO PAGE, LAST PAGE, NEXT PAGE, PREVIOUS PAGE.

GET FORM OBJECTS (objectsArray{; variablesArray{; pagesArray}}{; *})

Parameter	Type		Description
objectsArray	Array string	←	Name of form objects
variablesArray	Array pointer	←	Pointers to variables or fields associated with objects
pagesArray	Array integer	←	Page number of each object
*	*	→	If passed = reduce to the current page

Description

The GET FORM 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. 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.
Objects coming from an inherited form are considered as belonging to page 0 of the current page.

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.

See Also

GET FORM PROPERTIES.

GET FORM PARAMETER

version 2004.4

GET FORM 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 GET FORM 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 GET FORM 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 GET FORM 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.

See Also

Get database parameter, GET FORM PROPERTIES, SET DATABASE PARAMETER.

GET FORM PROPERTIES ({table; }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 GET FORM 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. If no name was defined, the title parameter returns an empty string.

See Also

GET FORM OBJECTS, Open form window, SET FORM SIZE.

GOTO PAGE (pageNumber)

Parameter	Type		Description
pageNumber	Number	→	Form page to display

Description

GOTO PAGE changes the currently displayed form page to the form page specified by pageNumber.

If no form is displayed or if pageNumber corresponds to the current page of the 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.

About form page management commands

Automatic action buttons perform the same tasks as the FIRST PAGE, LAST PAGE, NEXT PAGE, PREVIOUS PAGE and 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 Current form page command to find out which page is being displayed.

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:

```
GOTO PAGE (3)
```

See Also

Current form page, FIRST PAGE, LAST PAGE, NEXT PAGE, PREVIOUS PAGE.

INPUT FORM ({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 INPUT FORM 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.

INPUT FORM 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 INPUT FORM 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, INPUT FORM changes the input form for the table.

Examples

1. The following example shows a typical use of INPUT FORM:

```
INPUT FORM ([Companies]; "New Comp") ` Form for adding new companies
ADD RECORD ([Companies]) ` Add a new company
```

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")
    INPUT FORM([Invoices];"Input";"4D_SAS")
: (company="4D Inc")
    INPUT FORM([Invoices];"Input";"4D_Inc")
```

```
: (company="Acme")
    INPUT FORM([Invoices];"Input";"ACME")
End case
ADD RECORD([Factures])
```

See Also

ADD RECORD, CREATE USER FORM, DISPLAY RECORD, DISPLAY SELECTION, IMPORT DIF, IMPORT SYLK, IMPORT TEXT, MODIFY RECORD, MODIFY SELECTION, Open window, OUTPUT FORM, QUERY BY EXAMPLE.

LAST PAGE

Parameter	Type	Description

This command does not require any parameters

Description

LAST PAGE changes the currently displayed form page to the last form page. If a form is not being displayed, or if the last form page is already displayed, LAST PAGE does nothing.

Example

The following example is a one-line method called from a menu command. It displays the last form page:

```
LAST PAGE
```

See Also

Current form page, FIRST PAGE, GOTO PAGE, NEXT PAGE, PREVIOUS PAGE.

NEXT PAGE

Parameter	Type	Description

This command does not require any parameters

Description

NEXT PAGE changes the currently displayed form page to the next form page. If a form is not being displayed, or if the last form page is already displayed, 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:

```
NEXT PAGE
```

See Also

Current form page, FIRST PAGE, GOTO PAGE, LAST PAGE, PREVIOUS PAGE.

OUTPUT FORM ({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 OUTPUT FORM 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.

OUTPUT FORM 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 OUTPUT FORM 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 OUTPUT FORM. Note that although the OUTPUT FORM 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:

```
INPUT FORM ([Parts]; "Parts In") ` Select the input form
OUTPUT FORM ([Parts]; "Parts List") ` Select the output form
MODIFY SELECTION ([Parts]) ` This command uses both forms
```

See Also

CREATE USER FORM, DISPLAY SELECTION, EXPORT DIF, EXPORT SYLK, EXPORT TEXT, INPUT FORM, MODIFY SELECTION, PRINT LABEL, PRINT SELECTION.

PREVIOUS PAGE

Parameter	Type	Description
This command does not require any parameters		

Description

PREVIOUS PAGE changes the currently displayed form page to the previous form page. If a form is not being displayed, or if the first form page is already displayed, 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:

```
PREVIOUS PAGE
```

See Also

Current form page, FIRST PAGE, GOTO PAGE, LAST PAGE, NEXT PAGE.

SET FORM 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 SET FORM 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 SET FORM SIZE command.

See Also

SET FORM SIZE, SET FORM VERTICAL RESIZING.

SET FORM SIZE ({object; }horizontal; vertical{; *})

Parameter	Type		Description
object	String	→	Object name indicating form limits
horizontal	Longint	→	If * passed: horizontal margin (pixels) If * omitted: width (pixels)
vertical	Longint	→	If * passed: vertical margin (pixels) If * omitted: height (pixels
*	*	→	• If passed: add margins defined by horizontal and vertical parameters (automatic size or size based on object, if object is passed) • If omitted: use horizontal and vertical as width and height of the form

Description

The SET FORM 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:

SET FORM 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:

SET FORM 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.

• **Fixed size**
In you want to have a fixed form size, you must use the following syntax:

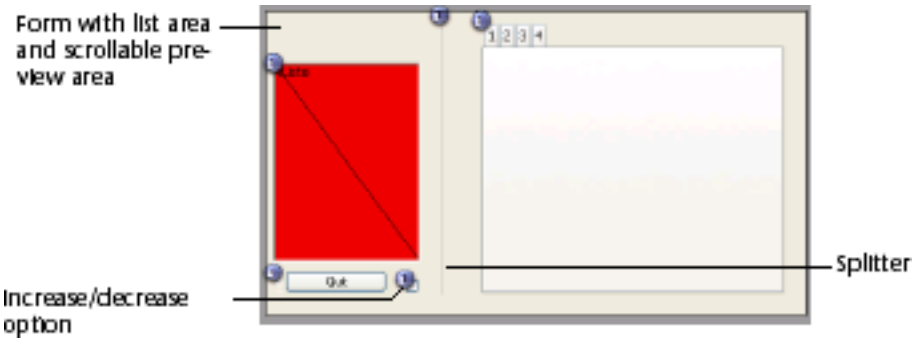
SET FORM SIZE(horizontal; vertical)

In this case, you must pass the width and height (in pixels) of the form in horizontal and vertical.

The SET FORM 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 :

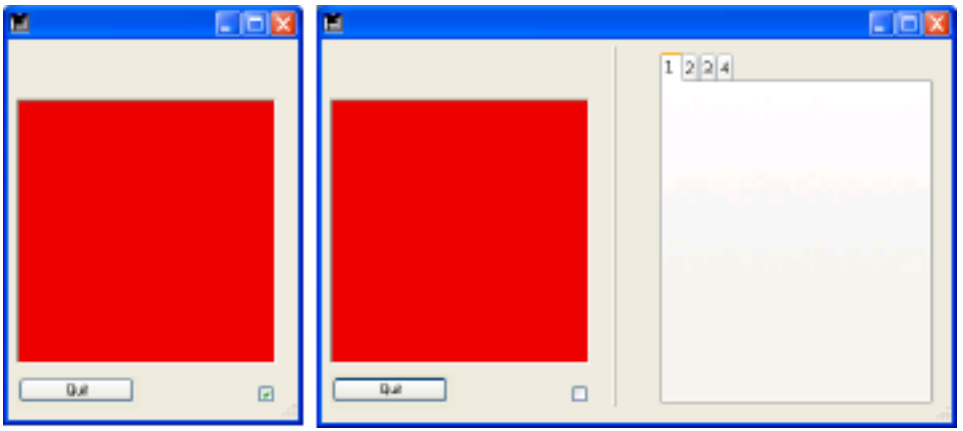


The size of the form is “automatic”.

The window is displayed using the following code:

```
$ref:=Open form window([Table 1];"Form1";Standard 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)
        SET FORM HORIZONTAL RESIZING(False)
        SET FORM SIZE("b1";margin;margin)
    Else
        SET FORM HORIZONTAL RESIZING(True)
        SET FORM SIZE("tab";margin;margin)
    End if

: (Form event=On click)
    <>collapsed:=b1
    If (b1)
            `collapsed
        GET OBJECT RECT(*;"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
            GET OBJECT RECT(*;"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)
            SET FORM HORIZONTAL RESIZING(True)
            SET FORM SIZE("tab";margin;margin)
         End if

      End case
```

See Also

SET FORM HORIZONTAL RESIZING, SET FORM VERTICAL RESIZING.

SET FORM VERTICAL RESIZING (resize{; minHeight{; maxHeight}})

Parameter	Type		Description
resize	Boolean	→	True: The form can be resized vertically False: The form cannot be resized vertically
minHeight	Longint	→	Smallest form height allowed (pixels)
maxHeight	Longint	→	Largest form height allowed (pixels)

Description

The SET FORM 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 SET FORM SIZE command.

See Also

SET FORM HORIZONTAL RESIZING, SET FORM SIZE.

19

Formulas

EDIT FORMULA (table; 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.

Note: 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.

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
```

See Also

APPLY TO SELECTION, EXECUTE FORMULA, SET ALLOWED METHODS.

System Variables or 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 (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.

The statement can be in the following:
- 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.

Example

See examples for the Command Name command.

See Also

Command name, EDIT FORMULA.

GET ALLOWED METHODS (methodsArray)

Parameter	Type	Description
methodsArray	Array string ←	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)
```

See Also

SET ALLOWED METHODS.

SET ALLOWED METHODS (methodsArray)

Parameter	Type	Description
methodsArray	Array string →	Array of method names

Description

The SET ALLOWED METHODS command allows you to define 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: Starting with version 2004.4 of 4D, the mechanism for restricting access to commands and methods in the Formula editor can be disabled for all users (compatibility option) or for the Designer and Administrator via two corresponding options in the Preferences. If the compatibility 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)
```

See Also

EDIT FORMULA, GET ALLOWED METHODS.

20

Graphs

GRAPH (graphArea; graphNumber; xLabels; yElements{; yElements2; ...; yElementsN})

Parameter	Type		Description
graphArea	Graph variable | Pict variable	→	Graph area or Picture variable
graphNumber	Number	→	Graph type number
xLabels	Array	→	Labels for the x-axis
yElements	Array	→	Data to graph (up to eight allowed)

Description

GRAPH draws a graph for a Graph area or a picture variable located in a form on the basis of values coming from arrays. The GRAPH command must be placed in the form method or in an object method belonging to the form, or yet again in a project method called by one of these two methods.

The graphs generated by this command can be drawn either using the integrated 4D Chart plug-in, or, beginning with 4D version 11, via the integrated SVG rendering engine.

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 DOM EXPORT TO PICTURE command can also be used to take advantage of the integrated SVG engine.

The type of the graphArea parameter determines which graphics engine is used for rendering: if you pass a 4D Chart area reference or a graph area variable, the 4D Chart plug-in will be used. If you pass a picture varaible, the SVG engine will be used. You can choose the type of engine to be used according to the following criteria:
• Graphs generated by 4D Chart can be entirely controlled, handled and enhanced by programming using the commands of the 4D Chart plug-in. For more information about 4D Chart commands, please refer to the 4D Chart *Language Reference* manual.
• Graphs generated by the SVG engine have a more modern appearance and benefit from interface functions associated with picture variables: contextual menu in Application mode (which can be used more particularly to choose the display format), scrollbars, etc.

In the graphArea parameter, pass either the graph area name (or a 4D Chart area reference), or a 4D picture variable, according to the rendering engine to be used. These areas are created in the Form editor in Design mode. For more information, see the *4D Design Reference* manual.

The graphNum parameter defines the type of graph that will be drawn. It must be 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 graphNum and executing the GRAPH command again.

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.

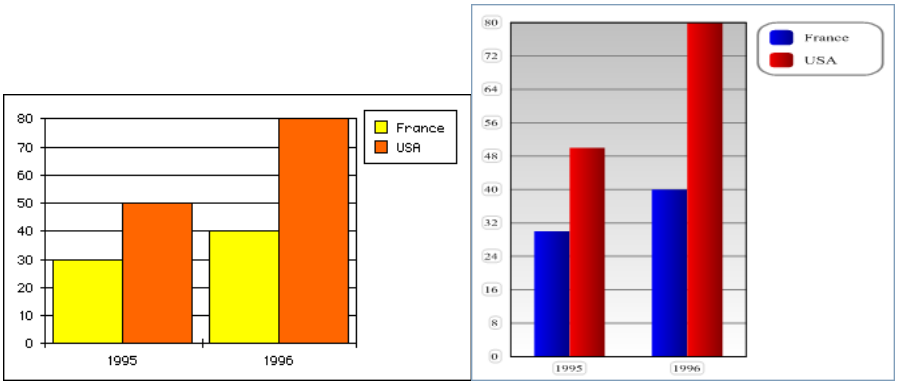
Example

The following example shows the different types of graphs that you can obtain with each graphics engine.. The code would be inserted in a form method or object method. It is not intended to be realistic, since the data is constant:

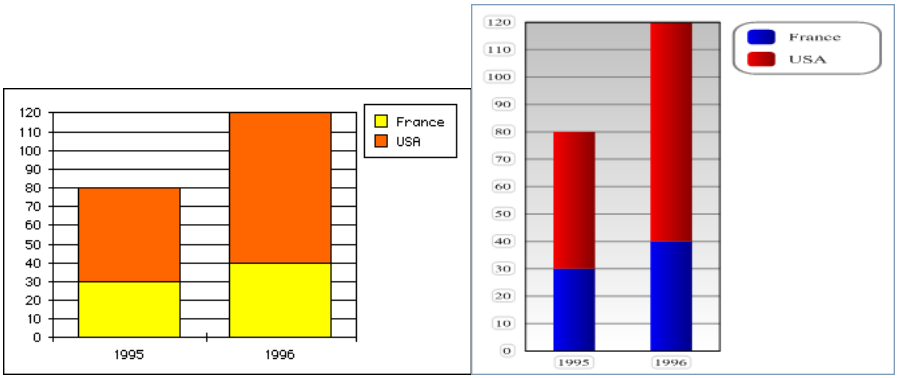
```
C_PICTURE (vGraph))  `Do not pass if you want to use the SVG engine
ARRAY STRING (4; X; 2)  ` Create an array for the x-axis
X{1}:="1995"  ` X Label #1
X{2}:="1996"  ` X Label #2
ARRAY REAL (A; 2)  ` Create an array for the y-axis
A{1}:=30  ` Insert some data
A{2}:=40
ARRAY REAL (B; 2) ` Create an array for the y-axis
B{1}:=50  ` Insert some data
B{2}:=80
GRAPH (vGraph;vType; X; A; B) ` Draw the graph
    ` Set the legends for the graph
GRAPH SETTINGS (vGraph;0;0;0;0;False;False;True;"France";"USA")
```

The following figure shows the resulting graph with each rendering engine (4D Chart then SVG).

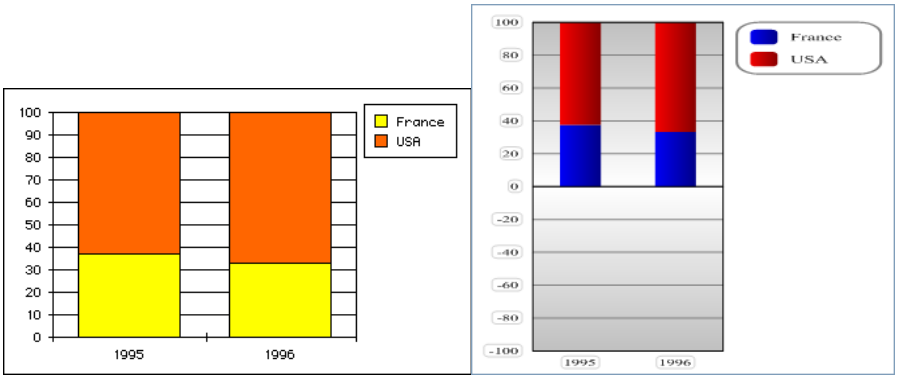
• With vType equal to 1, you obtain a **Column** graph:



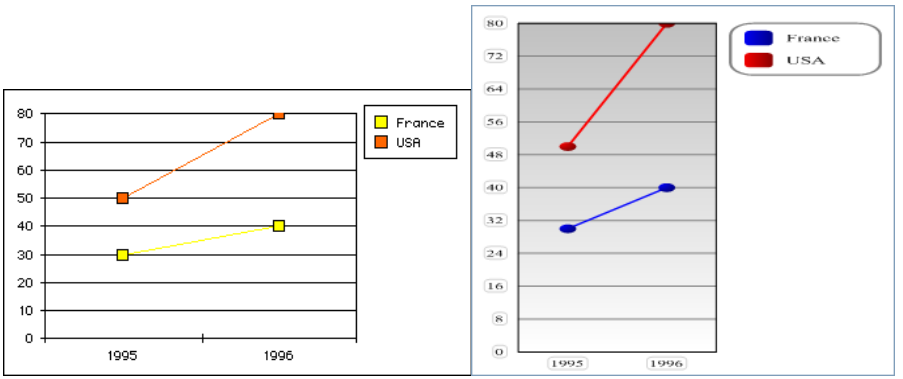
• With vType equal to 2, you obtain a **Proportional Column** graph:



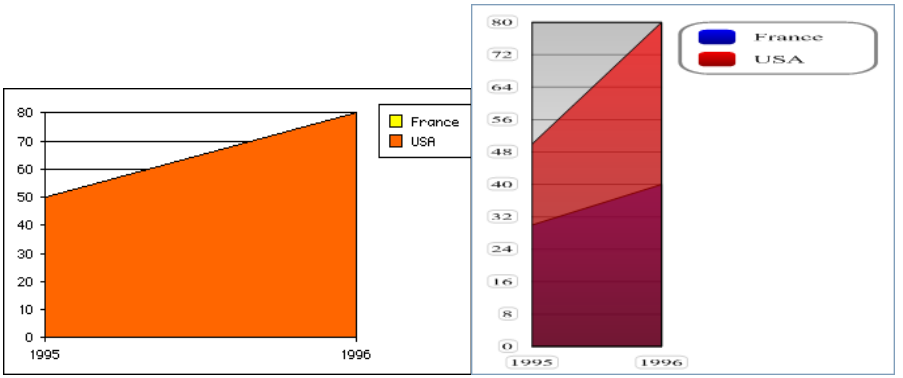
• With vType equal to 3, you obtain a **Stacked Column** graph:



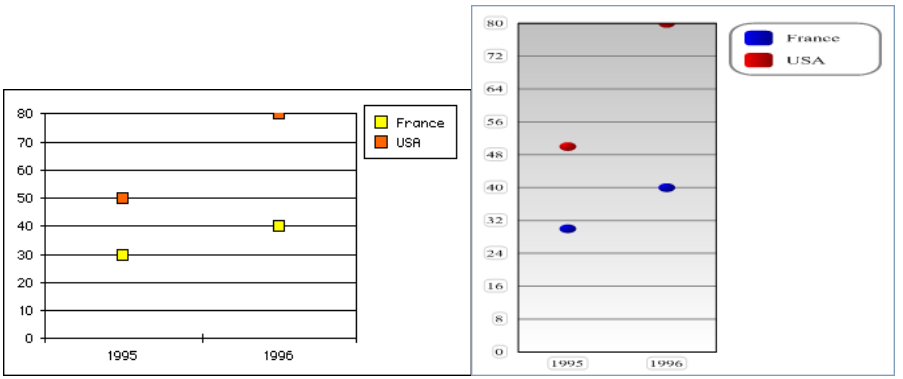
• With vType equal to 4, you obtain a **Line** graph:



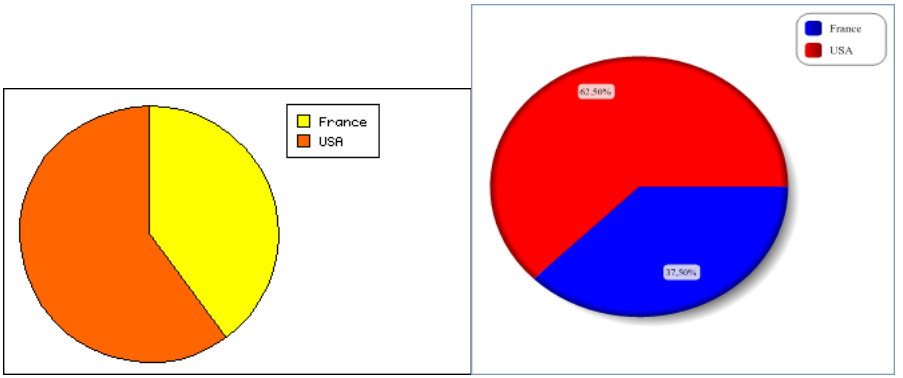
• With vType equal to 5, you obtain a **Area** graph:



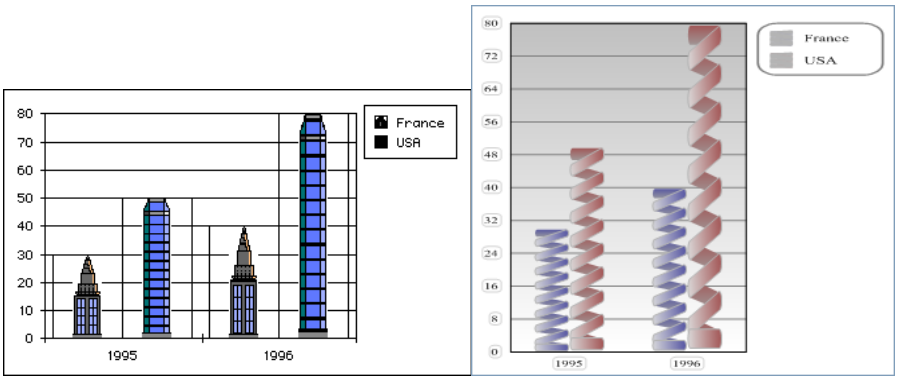
• With vType equal to 6, you obtain a **Scatter** graph:



• With vType equal to 7, you obtain a **Pie** graph:



• With vType equal to 8, you obtain a **Picture** graph:



See Also

CT Chart arrays, CT Chart data, CT Chart selection, DOM EXPORT TO PICTURE, GRAPH SETTINGS, GRAPH TABLE.

GRAPH SETTINGS (graph; xmin; xmax; ymin; ymax; xprop; xgrid; ygrid; title{; title2; ...; titleN})

Parameter	Type		Description
graph	Graph variable | Pict variable	→	Graph area or Picture variable
xmin	Number or date or time	→	Minimum x-axis value for proportional graph (line or scatter plot only)
xmax	Number or date or time	→	Maximum x-axis value for proportional graph (line or scatter plot only)
ymin	Number	→	Minimum y-axis value
ymax	Number	→	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 displayed with the GRAPH command. It can have been drawn using the 4D Chart plug-in (graph is a graph variable or a 4D Chart area reference) or by the SVG engine (graph is a picture variable). GRAPH SETTINGS has no effect on a pie chart. This command must be placed in the form method or in an object method belonging to the form, or yet again in a project method called by one of these two methods.

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 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.

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.

See Also

GRAPH, GRAPH TABLE.

GRAPH TABLE {(table)}

or:

GRAPH TABLE ({aTable; }graphType; x field; y field{; y field2; ...; y fieldN})

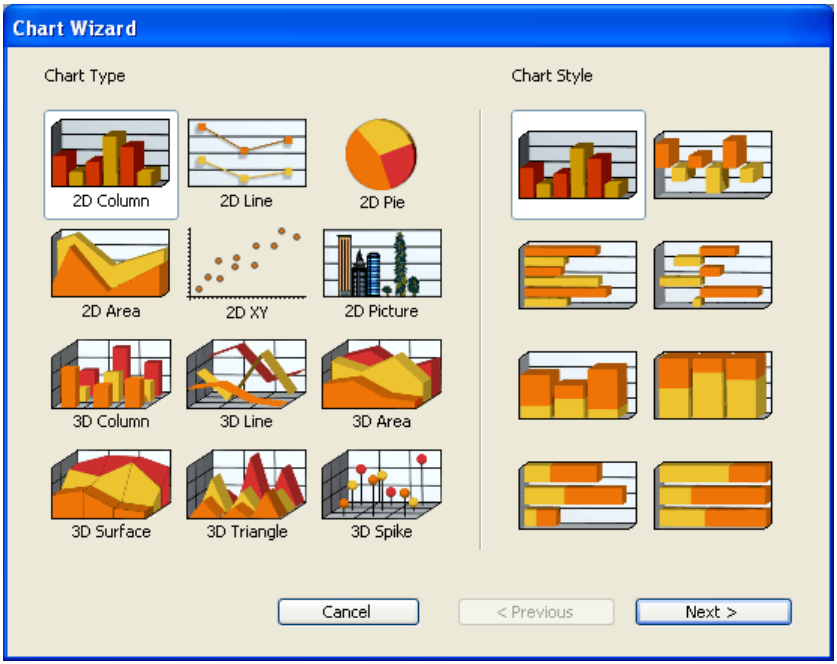
Parameter	Type		Description
aTable	Table	→	Table to graph, or Default table, if omitted
graphType	Number	→	Graph type number
x field	Field	→	Labels for the x-axis
y field	Field	→	Fields to graph (up to eight allowed)

Description

GRAPH TABLE has two forms. The first form displays the Chart Wizard of 4D Chart and allows the user to select the fields to be graphed. The second form specifies the fields to be graphed and does not display the Chart Wizard.

GRAPH TABLE graphs data from a table's fields. Only data from the current selection of the current process is graphed.

Using the first form is equivalent to choosing **Charts** from the **Tools** menu in the Design environment. The following figure shows the Chart Wizard, which allows the user to define the graph.



The second form of the command graphs the fields specified for aTable.

The graphType parameter defines the type of graph that will be drawn. It must be a number from 1 to 8. See the 4D Chart graph types listed in the example for the GRAPH command.

Note: The GRAPH TABLE command does not allow the use of the 4D SVG rendering engine.

The x field defines the labels that will be used to label the x-axis (the bottom of the graph). The field type can be Alpha, Integer, Long integer, Real or Date.

The y field is the data to graph. The field type must be Integer, Long integer or Real. Up to eight y fields can be graphed, each set off by a semicolon.

In either form, GRAPH TABLE opens a Chart window for working with the newly created graph. For more information about using the Chart window, see the *4D Design Reference* manual.

Note: You can also use the Quick Report editor to generate graphs from field data, by using the Print Destination menu.

Examples

1. The following example illustrates the use of the first form of GRAPH TABLE. It presents the Chart Wizard window and allows users to select the fields to graph. The code queries records in the [People] table, sorts them, and then displays the Chart Wizard:

```
QUERY ([People])
If (OK=1)
    ORDER BY ([People])
    If (OK=1)
        GRAPH TABLE([People])
    End if
End if
```

2. The following example illustrates the use of the second form of GRAPH TABLE. It first queries and orders records from the [People] table. It then graphs the salaries of the people:

```
QUERY([People];[People]Title="Manager")
ORDER BY([People];[People]Salary;>)
GRAPH TABLE([People];1;[People]Last Name;[People]Salary)
```

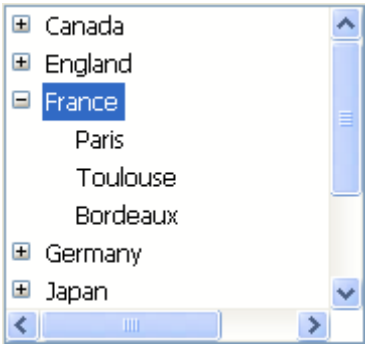
See Also

CT Chart selection, GRAPH.

21

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).

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 hierarhical 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. Except for SCROLL LINES, all of these commands belong to the "Object Properties" theme:
FONT
FONT STYLE
FONT SIZE
SET COLOR
SET FILTER
SET ENTERABLE
SET SCROLLBAR VISIBLE
SCROLL LINES ("User Interface" theme)
SET RGB COLORS

Reminder: Except for the SCROLL LINES 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 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 New 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

version 11 (Modified)

APPEND TO LIST (list; itemText; itemRef{; sublist; expanded})

Parameter	Type		Description
list	ListRef	→	List reference number
itemText	String	→	Text of the new list item (max. 255 characters)
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.

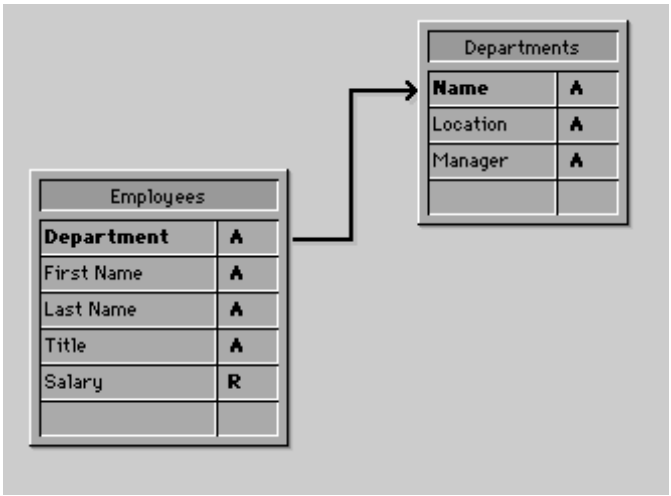
The list reference you pass in sublist must refer to an existing list. The existing list may be a one-level list or a list with sublists. If you do not want to attach a child list to the new item, omit the parameter or pass 0. Even though they are both optional, the sublist and expanded parameters must be passed jointly.

Tips

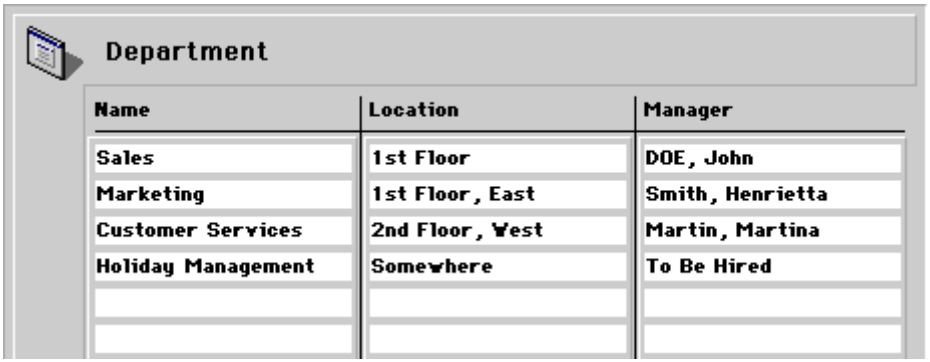
- To insert a new item in a list, use INSERT IN LIST. To change the text of an existing item or modify its child list as well as its expanded state, use SET LIST ITEM.
- To change the appearance of the new appended item use SET LIST ITEM PROPERTIES.

Example

Here is a partial view of a database structure:

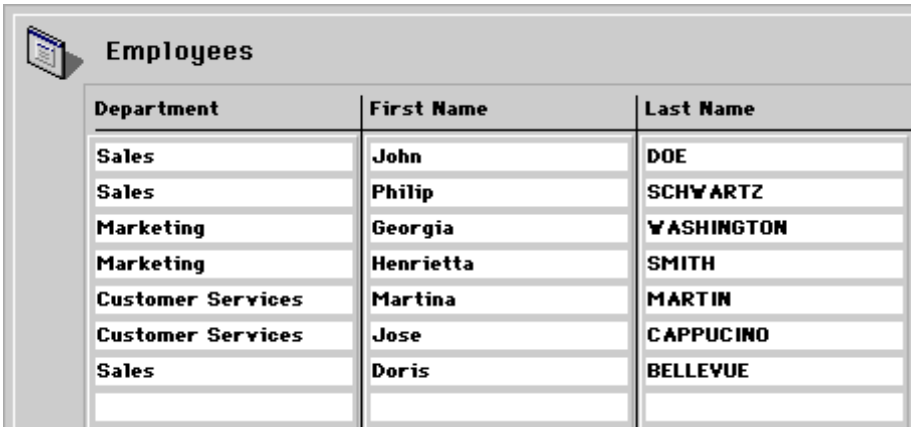


The [Departments] and [Employees] tables contain the following records:



Department

Name	Location	Manager
Sales	1st Floor	DOE, John
Marketing	1st Floor, East	Smith, Henrietta
Customer Services	2nd Floor, West	Martin, Martina
Holiday Management	Somewhere	To Be Hired



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)
      ` 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 the record number of the [Employees] record
            ` is passed as item reference number
         APPEND TO LIST($hSubList;[Employees]Last Name+", "+
                        [Employees]First Name;Record number([Employees]))
            ` Go the next [Employees] record
         NEXT RECORD([Employees])
      End for
   Else
         ` No Employees, no child list for the Department item
      $hSubList:=0
   End if
      ` Add the Department item to the main list
      ` Note that the record number of the [Departments] record
      ` is passed as item reference number. The bit #31
      ` of the item reference number is forced to one so we'll be able
      ` to distinguish Department and Employee items. See note further
      ` below on why we can use this bit as supplementary information about
      ` the item.
   APPEND TO LIST(hlList;[Departments]Name;0x80000000 |
                  Record number([Departments]);$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)+
                                                            $vsItemText+Char(34)+".")
            Else
                    ` If not, it is a double-click on an Employee item
                    ` Using the parent item reference number find the
                    `[Departments] record
                GOTO RECORD([Departments];List item parent(hlList;$vlItemRef)?-31)
                    ` 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
    ` The record number fits on 24 bits. In our example, we use bit #31 of the unused high
    ` byte for distinguishing Employees and Departments items.
```

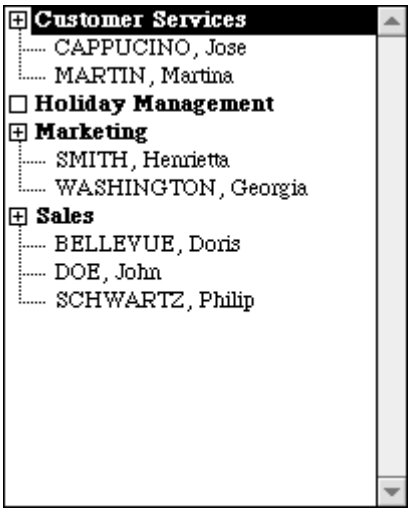
In this example, there is only one reason to distinguish [Departments] items and [Employees] items:

1. We store record numbers 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 parent item command to retrieve the parent of the selected item. If we click on an [Employees] item whose associated record number is #10, and if there is also a [Departments] item #10, the [Departments] item will be found first by List parent item 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.

See Also

INSERT IN LIST, SET LIST ITEM, SET LIST ITEM PROPERTIES.

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.

Examples

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
```

2. See example for the Load list command.
3. See example for the BLOB to list command.

See Also

BLOB to list, Load list, New list.

Copy list

Copy list (list) → ListRef

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.

See Also

CLEAR LIST, Load list, New list.

Count list items ({*; }list{; *}) → Longint

Parameter	Type		Description
*	*	→	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)
*	*	→	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.

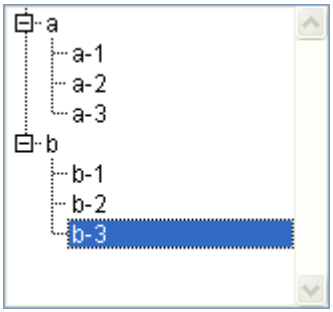
Note: If you use the @ character in the name of the list object and the form contains several lists that match with this name, the Count list items command will only apply to the first object whose name corresponds.

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.

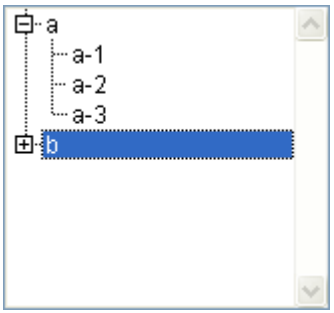
Examples

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
```

See Also

List item position, Selected list items.

DELETE FROM LIST ({*; }list; itemRef | *{; *})

Parameter	Type		Description
*	*	→	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, 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;*;*)
```

See Also

CLEAR LIST, GET LIST ITEM.

Find in list ({*; }list; value; scope{; itemsArray{; *}}) → Longint

Parameter	Type		Description
*	*	→	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
*	*	→	- 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 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 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.

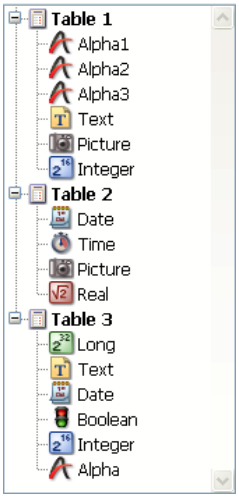
If you want to find out the position or number of all the items corresponding to value, pass a longint array in the optional itemsArray parameter. If necessary, the array will be created and resized by the command. The command will fill in the array with the positions (if the second * is omitted) or 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
$vlItemRef:Find in list(hList;"Date";1;$arrRefs;*)
    `$vlItemRef equals 11
    `$arrRefs{1} equals 11 and $arrRefs{2} equals 23
$vlItemPos:=Find in list(hList;"Date";0;*)
    `$vlItemPos equals 0
```

GET LIST ITEM ({*; }list; itemPos | *; itemRef; itemText{; sublist{; expanded}})

Parameter	Type		Description
*	*	→	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 | *	Number | *	→	Position of item in expanded 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/collaped 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.

Examples

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
```

2. Refer to the example of the APPEND TO LIST command.

See Also

GET LIST ITEM PROPERTIES, List item parent, List item position, Selected list items, SET LIST ITEM, SET LIST ITEM PROPERTIES.

Get list item font ({*; }list; itemRef | *) → String

Parameter	Type		Description
*	*	→	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 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.

See Also

SET LIST ITEM FONT.

GET LIST ITEM ICON ({*; }list; itemRef | *; icon)

Parameter	Type		Description
*	*	→	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 or 0 for the last item added to the list or * for the current item of the list
icon	Picture var	←	Icon associated with item

Description

The GET LIST ITEM ICON command returns, in icon, the icon associated with the item whose eference 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.

See Also

GET LIST ITEM PROPERTIES, SET LIST ITEM ICON.

GET LIST ITEM PARAMETER

GET LIST ITEM PARAMETER ({*; }list; itemRef | *; selector; value)

Parameter	Type		Description
*	*	→	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 or 0 for the last item appended to the list or * for the current list item
selector	String	→	Parameter constant
value	String | Boolean | Num	←	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 will be 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.

See Also

SET LIST ITEM PARAMETER.

GET LIST ITEM PROPERTIES ({*; }list; itemRef | *; enterable{; styles{; icon{; color}}})

Parameter	Type		Description
*	*	→	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, or 0 for last list item added, or * for the current list item
enterable	Boolean	←	TRUE = Enterable, FALSE = Non-enterable
styles	Number	←	Font style for the item
icon	Number	←	'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 parameter 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 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.

See Also

GET LIST ITEM, GET LIST ITEM ICON, SET LIST ITEM, SET LIST ITEM PROPERTIES.

GET LIST PROPERTIES (list; appearance{; icon{; lineHeight{; doubleClick{; multiSelections{; editable}}}}})

Parameter	Type		Description
list	ListRef	→	List reference number
appearance	Number	←	Graphical style of the list 1 = Hierarchical list a la Macintosh 2 = Hierarchical list a la Windows
icon	Number	←	'cicn' Mac OS-based resource ID
lineHeight	Number	←	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.

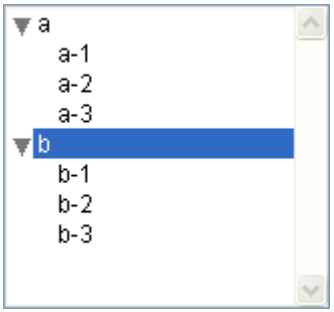
The parameter appearance returns the graphical style of the list.
The parameter icon returns the resource IDs of the node icons displayed in the list.
The parameter lineHeight returns the minimal line height.
If doubleClick is set to 1, double-clicking on a parent list item does not provoke its child 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 command SET LIST PROPERTIES and/or in the Design environment List Editor, if the list was created there or saved using the command SAVE LIST.

For a complete description of the appearance, node icons, minimal line height and double-click management of a list, see the command SET LIST PROPERTIES.

Example

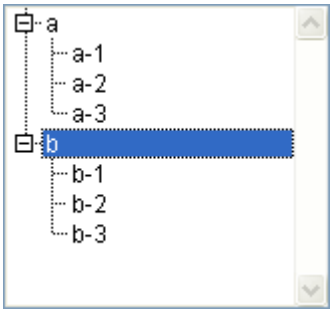
Given the list named hList, shown here in the Application environment:



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:



See Also

SET LIST PROPERTIES.

INSERT IN LIST({*; } list; beforeItemRef | *; itemText; itemRef{; sublist; expanded})

Parameter	Type		Description
*	*	→	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 | *	→	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 FROM LIST(hList;*;"New Item";vlUniqueRef)
```

See Also

APPEND TO LIST.

Is a list

Is a list (list) → Boolean

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.

Examples

1. See example for the command CLEAR LIST.
2. See examples for the command DRAG AND DROP PROPERTIES.

See Also

DRAG AND DROP PROPERTIES.

List item parent

version 11 (Modified)

List item parent ({*; }list; itemRef | *) → Longint

Parameter	Type		Description
*	*	→	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 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.

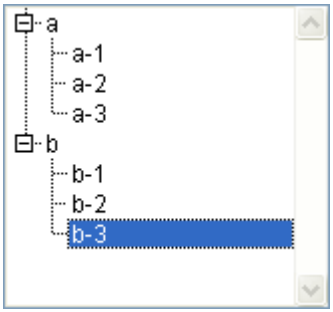
In return, if the corresponding item exists in the list and if this item is in a sublist (and therefore has a parent item), you obtain the item reference number of the parent item.

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.

Examples

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.

See Also

GET LIST ITEM, List item position, SELECT LIST ITEMS BY REFERENCE, SET LIST ITEM.

List item position

version 11 (Modified)

List item position ({*; }list; itemRef) → Number

Parameter	Type		Description
*	*	→	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	Number	←	Item position in expanded 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.

See Also

Count list items, SELECT LIST ITEMS BY REFERENCE.

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 (listName) → ListRef

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 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
```

See Also

CLEAR LIST, Is a list, SAVE LIST.

New list → ListRef

Parameter	Type		Description
This command does not require any parameters			
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 command APPEND TO LIST.

See Also

APPEND TO LIST, BLOB to list, CLEAR LIST, Copy list, DELETE FROM LIST, INSERT IN LIST, LIST TO BLOB, Load list.

version 11 (Modified)

REDRAW LIST (list)

Parameter	Type		Description
list	ListRef	→	List reference number

Compatibility note: The REDRAW LIST command serves no purpose beginning with version 11 of 4D. All representations of hierarchical lists are now automatically redrawn. When it is called, this command does nothing.

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.

See Also

Load list.

SELECT LIST ITEMS BY POSITION

version 11 (Modified)

SELECT LIST ITEMS BY POSITION ({*; }list; itemPos{; positionsArray})

Parameter	Type		Description
*	*	→	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	Number	→	Position of item in expanded list(s)
positionsArray	Number array	→	Array of the positions in the expanded 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.

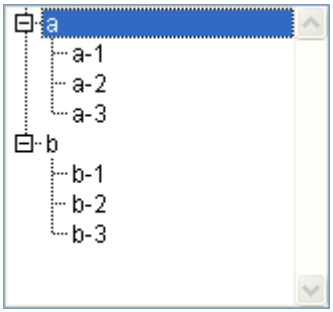
The optional positionsArray parameter lets you select several items simultaneously within the list. In positionsArray, you must pass an array where each line indicates the position of an item to be selected.

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.

Examples

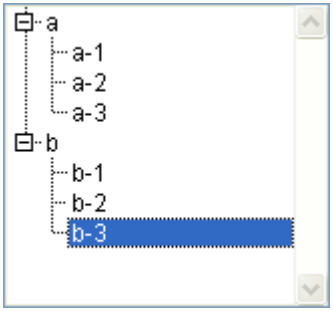
Given the hierarchical list named hList, shown here in the Application environment:



1. After the execution of this code:

```
SELECT LIST ITEMS BY POSITION(hList;Count list items(hList))
```

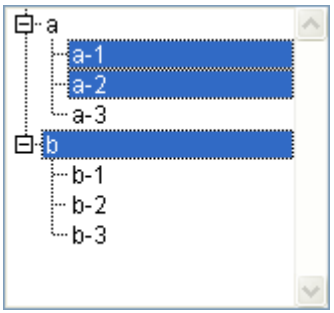
The last visible list item is selected:



2. 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:



See Also

EDIT ITEM, SELECT LIST ITEMS BY REFERENCE, Selected list items.

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:

```
$vlItemPos:=Selected list items(hList)  ` Get position of selected item
    ` Get item ref number of selected item
GET LIST ITEM(hList;$vlItemPos;$vlItemRef;$vsItemText)
    ` Get item ref. number of parent item (if any)
$vlParentItemRef:=List item parent(hList;$vlItemRef)
If ($vlParentItemRef>0)
        ` Select the parent item
    SELECT LIST ITEM BY REFERENCE(hList;List item parent(hList;$vlItemRef))
        ` Do NOT forget to call REDRAW LIST; otherwise the list won't be updated
End if
```

See Also

SELECT LIST ITEMS BY POSITION, Selected list items.

Selected list items

version 11 (Modified)

Selected list items ({*; }list{; itemsArray{; *}}) → Longint

Parameter	Type		Description
*	*	→	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 expanded list(s) If 2nd * passed: Array contains the selected item references
*	*	→	If omitted: Item position(s) If passed: Item reference(s)
Function result	Longint	←	If 2nd * omitted: Position of current selected list item in expanded 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 in the expanded 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 SELECT LIST ITEMS BY REFERENCE commands (programmed selection).

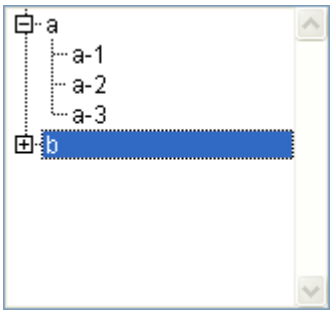
If the list has sublists, you apply the command to the main list (the one actually defined in the form), not one of its sublists. The positions are expressed relative to the top item of the main list, using the current expanded/collapsed state of the list and its sublist.

In any case, if no items are selected, the function returns 0.

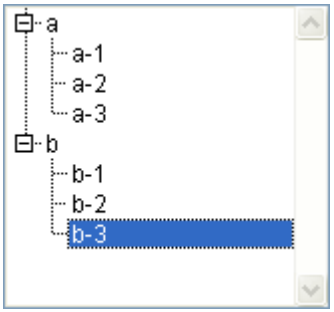
Examples

Here a list named hList, shown in the Application environment:

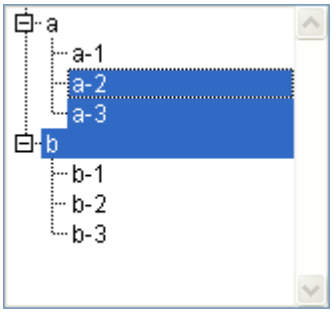
$vlItemPos:=**Selected list items**(hList) ` at this point $vlItemPos gets 2



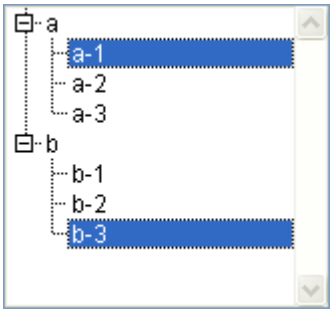
$vlItemPos:=**Selected list items**(hList) ` at this point $vlItemPos gets 4
$vlItemRef:=**Selected list items**(hList;*) ` $vlItemRef gets 200 (for instance)



$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)
```

See Also

SELECT LIST ITEMS BY POSITION, SELECT LIST ITEMS BY REFERENCE.

SET LIST ITEM (list; itemRef | *; newItemText; newItemRef{; sublist; expanded})

Parameter	Type		Description
*	*	→	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, 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 GET LIST ITEM, 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.

Examples

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
```

2. See example for the command GET LIST ITEM.
3. See example for the command APPEND TO LIST.

See Also

GET LIST ITEM, GET LIST ITEM PROPERTIES, SET LIST ITEM PROPERTIES.

SET LIST ITEM FONT ({*; }list; itemRef | *; font)

Parameter	Type		Description
*	*	→	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 or 0 for the last item added to the list or * for the current item of the list
font	String | Num	→	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")
```

See Also

FONT, Get list item font.

SET LIST ITEM ICON ({*; }list; itemRef | *; icon)

Parameter	Type		Description
*	*	→	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 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 can be used to modify 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: It is possible to modify the icon associated with an item using the SET LIST ITEM PROPERTIES command. However, SET LIST ITEM PROPERTIES 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 will be 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.

The use of pointers is particularly recommended since hierarchical lists are optimized in this case: only one instance of the picture will be created in memory if the same icon is used for several different list items.

Note: Conversely, the direct use of variables generated by the GET ICON RESOURCE or GET PICTURE RESOURCE commands is not recommended because the icon will be duplicated in memory for each item of the list.

Example

This code has been optimized thanks to the use of a pointer:

```
vIcon:=[Params]Icon
SET LIST ITEM ICON(mylist;ref1;vIcon->)
SET LIST ITEM ICON(mylist;ref2;vIcon->)
```

See Also

SET LIST ITEM PROPERTIES.

SET LIST ITEM PARAMETER ({*; }list; itemRef | *; selector; value)

Parameter	Type		Description
*	*	→	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 or 0 for the last item appended to the list or * for the current list item
selector	String	→	Parameter constant
value	String | Boolean | Num	→	Value of the parameter

Description

The SET LIST ITEM PARAMETER command can be used to modify 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 APPEND TO LIST).
Lastly, you can pass * in itemRef: in this case, the command will be 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.

See Also

GET LIST ITEM PARAMETER.

SET LIST ITEM PROPERTIES ({*; }list; itemRef | *; enterable; styles; icon{; color})

Parameter	Type		Description
*	*	→	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, or 0 for last item appended to the list, or * for the current list item
enterable	Boolean	→	TRUE = Enterable, FALSE = Non-enterable
styles	Number	→	Font style for the item
icon	Number	→	'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 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:

Constant	Type	Value
Plain	Long Integer	0
Bold	Long Integer	1
Italic	Long Integer	2
Underline	Long Integer	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 SET RGB COLORS command. Pass -1 in the color parameter to reset the original color of the item.

Examples

1. See the example for the command APPEND TO LIST.

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)
```

See Also

GET LIST ITEM PROPERTIES, SET LIST ITEM, SET LIST ITEM ICON.

SET LIST PROPERTIES (list; appearance{; icon{; lineHeight{; doubleClick{; multiSelections{; editable}}}}})

Parameter	Type		Description
list	ListRef	→	List reference number
appearance	Number	→	Graphical style of the list 1 = Hierarchical list a la Macintosh 2 = Hierarchical list a la Windows 0 = Auto appearance depending on platform
icon	Number	→	'cicn' Mac OS-based resource ID or 0 for default platform node icon
lineHeight	Number	→	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

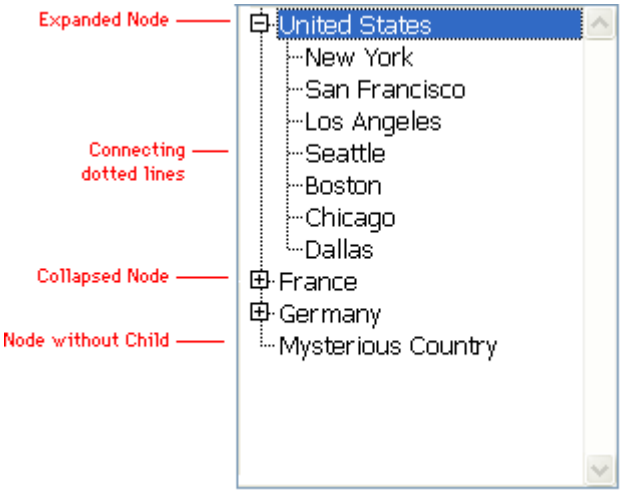
The SET LIST PROPERTIES command sets the appearance of the hierarchical list whose list reference you pass in list.

The parameter appearance can be one of the following predefined constants provided by 4D in the Hierarchical Lists theme:

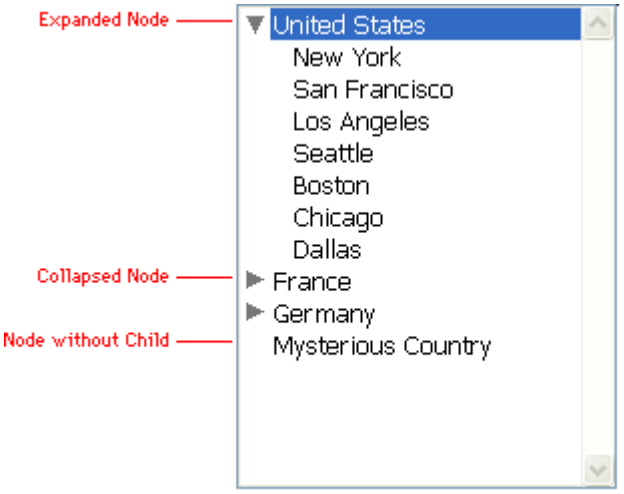
Constant	Type	Value
ala Macintosh	Long Integer	1
ala Windows	Long Integer	2

In the Windows appearance, the list has connecting dotted lines between the nodes and branches. One icon (+) denotes the collapsed nodes, a second one (–) the expanded nodes. Nodes without child items have no icon.

Here is a default hierarchical list in Windows appearance:



In the Macintosh appearance, the list has no connecting dotted lines. One icon denotes the collapsed nodes, a second one 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	Long Integer	860
Windows node	Long Integer	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 icons that is too tall or too wide, it will be displayed truncated and/or will be overidden by the connecting dotted lines (if appearance is Windows), as well as 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.

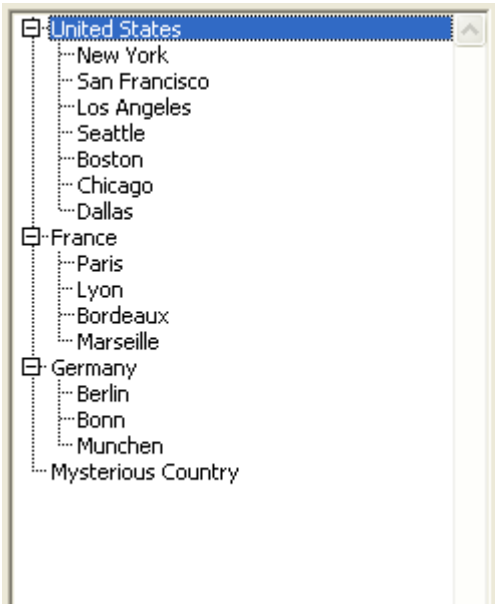
The optional parameter doubleClick allows you to define that a double-click on a parent list item will not provoke the sublist to expand or to collapse. By default, a double-click on a parent list item provokes its child list to expand or to collapse. However, some user interfaces may need to deactivate 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.

Examples

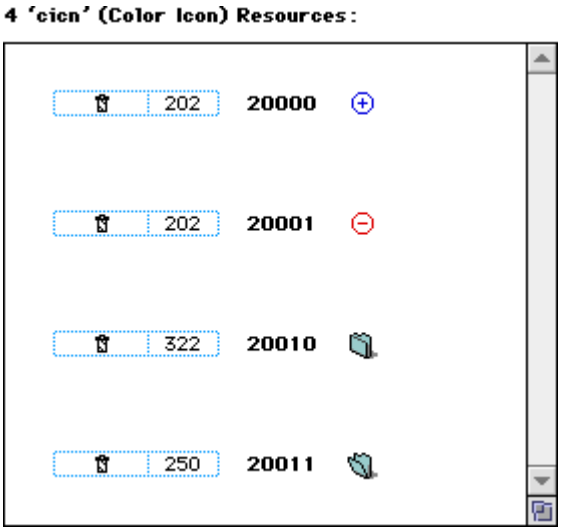
The following hierarchical list has been defined in the Design environment List Editor:



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
```

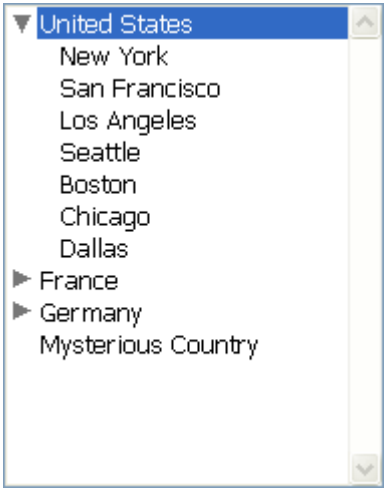
In addition, the structure file of the database has been edited so it contains the following 'cicn' color icon resources:



1. With the following line:

SET LIST PROPERTIES(hlCities;Ala Macintosh;Macintosh node)

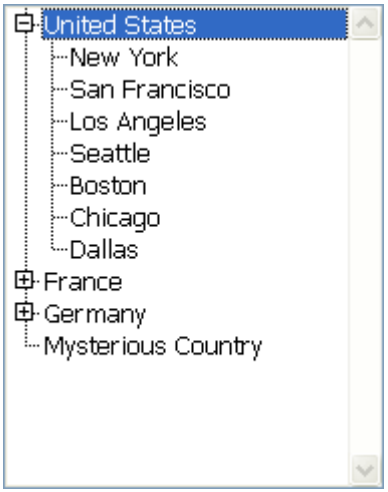
The hierarchical list will look like this:



2. With the following line:

SET LIST PROPERTIES(hlCities;Ala Windows;Windows node)

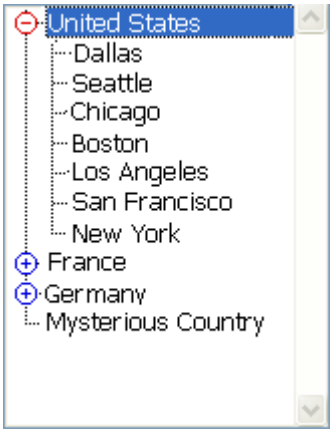
The hierarchical list will look like this:



3. With the following line:

SET LIST PROPERTIES(hlCities;Ala Windows;20000)

The hierarchical list will look like this:

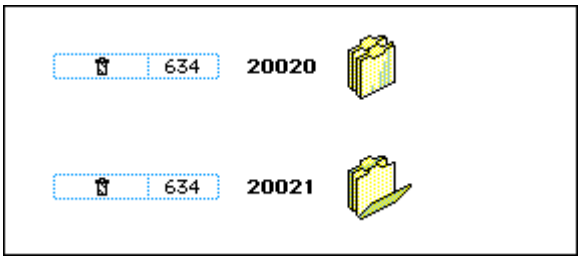


4. With the following line:

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:

See Also

GET LIST ITEM PROPERTIES, GET LIST PROPERTIES, SET LIST ITEM PROPERTIES.

SORT LIST (list{; > or <})

Parameter	Type		Description
list	ListRef	→	List reference number
> or <		→	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.

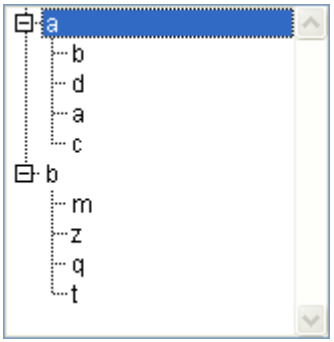
To sort in ascending order, pass >. To sort in descending order, pass <. If you omit the sorting order parameter, SORT LIST sorts in ascending order by default.

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

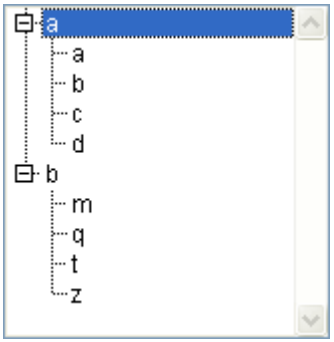
Given the list named hList, shown here in the Application environment (in Macintosh appearance):



After the execution of this code:

```
      ` Sort the list and it sublists in ascending order
SORT LIST(hList;>)
```

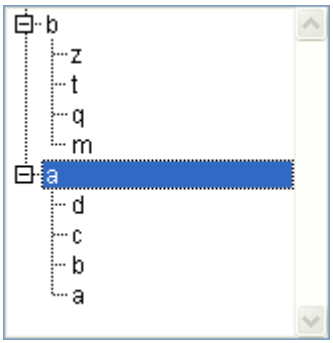
The list looks like:



After the execution of this code:

```
      ` Sort the list and it sublists in ascending order
SORT LIST(hList;<)
```

The list looks like:



See Also

Selected list items.

22

Import and Export

EXPORT DATA (fileName{; project{; *}})

Parameter	Type		Description
fileName	String	→	Full path name of the export file
project	BLOB	→	Contents of the export project
		←	New contents of the export project (if the * parameter has been passed)
*	*	→	Displays the export dialog box and updates the project

Description

The EXPORT DATA command allows you to export 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.

• If you don't pass the optional parameter project, the export dialog box is displayed. The user can define the export parameters or load an existing export project.

Note: An export project contains all the export parameters such as the tables and fields to export, delimiters, etc. You define these parameters in the export dialog box. A project can be saved to disk and then loaded. For more information about the export dialog box, please refer to the *4D Design Reference* manual.

• If you pass a BLOB containing a valid export project to the project parameter, the export will be directly performed, without the user intervening. The project must already be predefined in the export dialog box, then saved. To do so, you have two possible solutions:
- Save the project to disk, then load it by using the DOCUMENT TO BLOB command, in a field or a variable of type BLOB that you pass to the project parameter.

- Use the EXPORT DATA command with an empty project parameter and the optional parameter *, then store the project parameter in a field of type BLOB (see below). This solution allows you to save the project with the datafile without having to load it from a BLOB on disk.

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 BLOB field, on disk, etc.

If the export was successful, the OK system variable is equal to 1.

Example

This example creates an empty project and stores the parameters set by the user in the export dialog box there:

```
C_BLOB($exportParams)
SET BLOB SIZE($exportParams;0)  `Initialization of BLOB
EXPORT DATA("DocExport.txt";$exportParams;*)  ` Display of the export dialog box
```

See Also

EXPORT DIF, EXPORT SYLK, EXPORT TEXT, IMPORT DATA.

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 ({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.

In Unicode mode (standard mode), the command uses the default UTF-8 character set. You can use the USE CHARACTER SET command to change this character set.
In ASCII compatibility mode, the export operation is made using the default ASCII map for the platform on which it is executed, unless the command USE CHARACTER SET is used prior to the export.

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:

```
OUTPUT FORM([People];"Export")
EXPORT DIF([People];"NewPeople.dif"") ` Export to the "NewPeople.dif" document
```

See Also

EXPORT SYLK, EXPORT TEXT, IMPORT DIF, USE CHARACTER SET.

System Variables and Sets

OK is set to 1 if the export is successfully completed; otherwise, it is set to 0.

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.

In Unicode mode (standard mode), the command uses the default UTF-8 character set. You can use the USE CHARACTER SET command to change this character set.
In ASCII compatibility mode, the export operation is made using the default ASCII map for the platform on which it is executed, unless the command USE CHARACTER SET is used prior to the export.

When using EXPORT 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 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:

```
OUTPUT FORM([People];"Export")
EXPORT SYLK([People];"NewPeople.slk") ` Export to the "NewPeople.slk" document
```

See Also

EXPORT DIF, EXPORT TEXT, IMPORT SYLK, USE CHARACTER SET.

System Variables and Sets

OK is set to 1 if the export is successfully completed; otherwise, it is set to 0.

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.

In Unicode mode (standard mode), the command uses by default the UTF-8 character set. You can use the USE CHARACTER SET command to change this character set.
In ASCII compatibility mode, the export operation uses the ASCII table of the platform on which it is executed, except when the USE CHARACTER SET command was used prior to the export.

Using EXPORT 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 Export Data dialog box. Text fields may contain carriage returns, so be careful when using a carriage return as a delimiter if you are exporting text fields.

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:

```
OUTPUT FORM([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
```

See Also

EXPORT DIF, EXPORT SYLK, IMPORT TEXT, USE CHARACTER SET.

System Variables and Sets

OK is set to 1 if the export is successfully completed; otherwise, it is set to 0.

IMPORT DATA (fileName{; project{; *}})

Parameter	Type		Description
fileName	String	→	Access path and name of the import file
project	BLOB	→	Contents of the import project
		←	New contents of the import project (if the * parameter has been passed)
*	*	→	Displays the import dialog box and updates the project

Description

The IMPORT DATA command allows you to import 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.

• If you do not pass the optional parameter project, the import dialog box is displayed. The user can define then import parameters or load an existing import project.

Note: 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. Those parameters are defined in the import dialog box. An import project can be saved to disk to be loaded and used later. For more information about the import dialog box, please refer to the *4D Design Reference* manual.

• If you pass a BLOB containing a valid import project in the project parameter, the import will be directly performed and will not require the user's intervention. The project must already be predefined in the import dialog box, then saved. To do so, you have two possible solutions:
- Save the project to disk, then load it, using the DOCUMENT TO BLOB command, in a BLOB field or a BLOB variable that you pass in project.

- Use the IMPORT DATA command with an empty project parameter and the optional parameter *, then store the project parameter in a BLOB field (see below). This solution allows you to save the project with the datafile without having to load it from a BLOB located on the disk.

Note: Refer to the EXPORT DATA command for an example concerning the definition of an empty project.

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.

See Also

EXPORT DATA, IMPORT DIF, IMPORT SYLK, IMPORT TEXT.

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 ({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.

In Unicode mode (standard mode), the command uses the default UTF-8 character set. You can use the USE CHARACTER SET command to change this character set.
In ASCII compatibility mode, the import operation is made using the default ASCII map for the platform on which it is executed, unless the command USE CHARACTER SET is used prior to the import.

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:

```
INPUT FORM([People]; "Import")
IMPORT DIF([People];"NewPeople.dif") ` Import from “NewPeople.dif” document
```

See Also

EXPORT DIF, IMPORT SYLK, IMPORT TEXT, USE CHARACTER SET.

System Variables and Sets

OK is set to 1 if the import is successfully completed; otherwise, it is set to 0.

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.

In Unicode mode (standard mode), the command uses the default UTF-8 character set. You can use the USE CHARACTER SET command to change this character set.
In ASCII compatibility mode, the import operation is made using the default ASCII map for the platform on which it is executed, unless the command USE CHARACTER SET is used prior to the import.

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:

```
INPUT FORM([People]; "Import")
IMPORT SYLK([People];"NewPeople.slk") ` Import from “NewPeople.slk” document
```

See Also

EXPORT SYLK, IMPORT DIF, IMPORT TEXT, USE CHARACTER SET.

System Variables and Sets

OK is set to 1 if the import is successfully complete; otherwise, it is set to 0.

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.

In Unicode mode (standard mode), the command uses by default the UTF-8 character set. You can use the USE CHARACTER SET command to change this character set.
In ASCII compatibility mode, the import operation uses the ASCII table of the platform on which it is executed, except when the USE CHARACTER SET command was used prior to the import.

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:

```
INPUT FORM([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
```

See Also

EXPORT TEXT, IMPORT DIF, IMPORT SYLK, USE CHARACTER SET.

System Variables and Sets

OK is set to 1 if the import is successfully completed; otherwise, it is set to 0.

23

Interruptions

Note: You will rarely call this command.

ABORT

Parameter	Type	Description

This command does not require any parameters

Description

The ABORT command is to be 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.

See Also

ON ERR CALL.

FILTER EVENT

Parameter	Type	Description

This command 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 allows you to remove 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: Avoid creating an event-handling method that only calls the FILTER EVENT command, because all the events are going to be ignored by 4D. In case 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 command ON EVENT CALL.

See Also

ON EVENT CALL.

Method called on error → String

Parameter	Type		Description
This command does not require any parameters			
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)
```

See Also

ON ERR CALL.

Method called on event → String

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

ON EVENT 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**.

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.

To stop the trapping of errors, call ON ERR CALL again and pass the empty string in errorMethod.

Once an error-handling project is installed, 4D calls the method each time an error occurs.

You can identify errors by reading the Error system variable, which contains the code number of the error. Error codes are listed in the theme Error codes. For more information, see the section Syntax Errors or Database Engine Errors. 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.

The error-handling method should manage the error in an appropriate way or present an error message to the user. Errors can be generated by:
• The 4D database engine; for example, when saving a record tries to duplicate a unique index key.
• The 4D environment; for example, when you do not have enough memory for allocating an array.
• The operating system on which the database is runs; 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.

ON ERR CALL is usually placed in the startup database method of an application, in order to handle errors for this application. ON ERR CALL can also be placed at the start of a method in order to handle errors specific to this method.

When an ON ERR CALL error-handling method is installed, it is not possible to trace a method by using Alt+Click (on Windows) or Option-Click (on Macintosh). This is because Alt+Click and Option-Click) generate an error (error code 1006) that immediately activates the ON ERR CALL error-handling method. However, you can test this error code by calling TRACE.

Examples

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
```

2. See example in the section Arrays and Memory.

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
    ` ...
```

4. The following error-handling method ignores the user interruptions:

```
    ` SHOW ONLY ERRORS project method
If (Error#1006)
    ALERT ("The error "+String(Error)+" occurred.")
End if
```

See Also

ABORT, Method called on error.

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-Option-Control-Backspace (on Macintosh) converts the Event Manager process into a normal process. This means that the method will no longer be automatically passed all the events that occur. 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 ASCII Codes (and its subsections) and Function Key Codes. 4D provides predefined constants for the major ASCII Codes and for Function Key Codes. In the Explorer window, look for the themes of these constants.
• The Modifiers system variable contains the modifier value. It indicates whether any of the following modifier keys were down when the event occurred:

Platform	Modifiers
Windows	Shift key, Caps Lock, Alt key, Ctrl key, Right mouse button
Macintosh	Shift key, Caps Lock, Option key, Command key, Control key

Notes
- The Windows ALT key is equivalent to the Macintosh Option key.
- The Windows Ctrl key is equivalent to the Macintosh Command key.
- The Macintosh Control key has no equivalent on Windows. However, a right mouse button click on Windows is equivalent to a Control-Click on Macintosh.

The modifier keys do not generate an event; another key or the mouse button must also be pressed. The Modifiers variable is a 4-byte Long Integer variable that should be seen as an array of 32 bits. 4D provides predefined constants expressing bit positions or bit masks for testing the bit corresponding to each modifier key. For example, to detect if the Shift key was pressed for the event, you can write:

```
If (Modifiers ?? Shift key bit ) ` If the Shift key was down
```

or:

```
If ((Modifiers & Shift key mask)#0)` If the Shift key was down
```

Note: Under Windows, the value 128 is added to the Modifiers variable if the (left) button of the mouse is released at the time of the event.

• 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; PRINT SELECTION does that for you.

See Also

FILTER EVENT, GET MOUSE, Method called on event, Shift down.

24

Language

Command name (command) → String

Parameter	Type		Description
command	Number	→	Command number
Function result	String	←	Localized command name

Description

The Command name command returns the literal name of the command whose command number you pass in command.

4D integrates a dynamic translation of the keywords, constants, and command names used in your methods. For example, if you use the English version of 4D, you write:

```
DEFAULT TABLE ([MyTable])
ALL RECORDS ([MyTable])
```

This same code, reopened with the French version of 4D, will read:

```
TABLE PAR DEFAUT ([MyTable])
TOUT SELECTIONNER ([MyTable])
```

However, 4D also includes a unique feature, the EXECUTE FORMULA command, which allows you to build code on the fly and then execute this code, even though the database is compiled.

The example code, written with EXECUTE FORMULA statements in English, looks like:

```
EXECUTE FORMULA ( "DEFAULT TABLE([MyTable])")
EXECUTE FORMULA( "ALL RECORDS([MyTable])")
```

This same code, reopened with the French version of 4D, will then read:

```
EXECUTER FORMULE ( "DEFAULT TABLE([MyTable])")
EXECUTER FORMULE( "ALL RECORDS([MyTable])")
```

4D automatically translates EXECUTE FORMULA (English) to EXECUTER FORMULE (French), but cannot translate the text statement you passed to the command.

If you use the EXECUTE FORMULA command in your application, you can use Command name to eliminate international localization issues for statements you execute in this way, and thus make your statements independent of language. The example code becomes:

```
EXECUTE FORMULA (Command name (46)+"([MyTable])")
EXECUTE FORMULA(Command name (47)+"([MyTable])")
```

With a French version of 4D, this code will read:

```
EXECUTER FORMULE(Nom commande (46)+"([MyTable])")
EXECUTER FORMULE(Nom commande (47)+"([MyTable])")
```

Note: To know the number of a command, refer to the Command Syntax by Name section.

Examples

1. For all the tables of your database, you have a form called “INPUT FORM” used for standard data entry in each table. Then, you want to add a generic project method that will set this form as the current input form for the table whose pointer or name you pass. You write:

```
    ` STANDARD INPUT FORM project method
    ` STANDARD INPUT FORM ( Pointer {; String })
    ` STANDARD INPUT FORM ( ->Table {; TableName })
C_POINTER ($1)
C_STRING (31;$2)

If (Count parameters>=2)
    EXECUTE FORMULA(Command name (55)+"(["+$2+"];"INPUT FORM")")
Else
    If (Count parameters>=1)
        INPUT FORM ($1->;"INPUT FORM")
    End if
End if
```

After this project method has been added to your database, you write:

```
STANDARD INPUT FORM  (->[Employees])
STANDARD INPUT FORM  ("Employees")
```

Note: Usually, it is better to use pointers when writing generic routines. First, the code will run compiled if the database is compiled. Second, 4D Insider will retrieve the references to the object whose pointer you pass. Third, as in the previous example, your code can cease to work correctly if you rename the table. However, in certain cases, using EXECUTE FORMULA will solve the problem.

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) ` Sum
        asCommand{2}:=Command name (2) ` Average
        asCommand{3}:=Command name (4) ` Min
        asCommand{4}:=Command name (3) ` Max
    ` ...
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.

See Also

Command Syntax by Name, EXECUTE FORMULA.

Count parameters → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	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 Count parameters is associated with a menu, Count parameters returns 0.

Examples

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
```

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)
```

See Also

Compiler Commands.

Current method name → String

Parameter	Type		Description
This command does not require any parameters			
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
Form Method	[TableName]FormName
Object Method	[TableName]FormName.ObjectName

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 application Preferences) 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 (methodName; result | *{; param}{; param2; ...; paramN})

Parameter	Type		Description
methodName	String	→	Name of project method to be executed
result | *	Variable | *	←	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 causes the execution of 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.

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).

See Also

EXECUTE FORMULA.

System Variables or Sets

If this command is executed correctly, the system variable OK is set to 1; otherwise, it is set to 0.

Get pointer (varName) → Pointer

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}" to Get pointer. However, you cannot use 2D array elements (ArrName+"{3}{5}") or variable elements (ArrName+"{myVar}").

Example

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
```

See Also

Field, Table.

Is a variable (aPointer) → Boolean

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).

Starting with version 6, instead of using Is a variable, it will be more convenient to use RESOLVE POINTER, which tells you about the nature of the referenced object, no matter what the object is (including the case of Nil pointers).

See Also

Nil, RESOLVE POINTER.

Nil (aPointer) → Boolean

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).

Starting with version 6, instead of using Nil, it will be more convenient to use RESOLVE POINTER, which tells you about the nature of the referenced object, no matter what the object is (including Nil pointers).

See Also

Is a variable, RESOLVE POINTER.

NO TRACE

Parameter	Type	Description
This command 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.

See Also

TRACE.

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	Number	←	Number of referenced table or array element or 0 or -1
fieldNum	Number	←	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	0
Array	Name of the array	-1	0
Array element	Name of the array	Element number	0
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.

Examples

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).

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
` ...
```

3. See example for the DRAG AND DROP PROPERTIES command.

See Also

DRAG AND DROP PROPERTIES, Field, Get pointer, Is a variable, Nil, Table.

Self → Pointer

Parameter	Type		Description
This command does not require any parameters			
Function result	Pointer	←	Pointer to form object (if any) whose method is currently being executed. Otherwise Nil (->[]) if outside of context

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:
- 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.

Example

See the example for the RESOLVE POINTER command.

See Also

RESOLVE POINTER.

TRACE

Parameter	Type	Description

This command 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
```

See Also

NO TRACE.

Type (fieldVar) → Number

Parameter	Type		Description
fieldVar	Field | Variable	→	Field or Variable to be tested
Function result	Number	←	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:

Constant	Type	Value
Is Alpha Field	Long Integer	0
Is String Var	Long Integer	24
Is Text	Long Integer	2
Is Real	Long Integer	1
Is Integer	Long Integer	8
Is LongInt	Long Integer	9
Is Date	Long Integer	4
Is Time	Long Integer	11
Is Boolean	Long Integer	6
Is Picture	Long Integer	3
Is Subtable	Long Integer	7
Is BLOB	Long Integer	30
Is Undefined	Long Integer	5
Is Pointer	Long Integer	23
String array	Long Integer	21
Text array	Long Integer	18
Real array	Long Integer	14
Integer array	Long Integer	15
LongInt array	Long Integer	16
Date array	Long Integer	17

Boolean array	Long Integer	22
Picture array	Long Integer	19
Pointer array	Long Integer	20
Array 2D	Long Integer	13

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).

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).

Examples

1. See example for the APPEND DATA TO PASTEBOARD command.

2. See example for DRAG AND DROP PROPERTIES command.

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 [Things To Do] table except Alphanumeric fields
EMPTY RECORD (->[Things To Do]; -1 ?- Is Alpha Field )
```

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
```

See Also

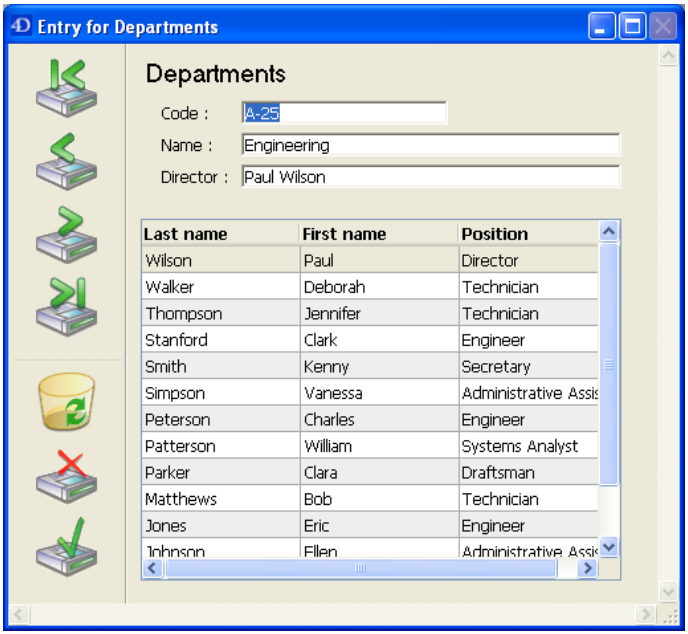
Is a variable, Undefined.

25

List Box

version 11 (Modified)

The commands of this theme are dedicated to handling form objects of the List box type. List boxes are comparable to Grouped Scrollable Areas. A list box provides all the functions of grouped scrollable areas, notably the ability to represent data in the form of columns and selectable rows. However, the list box does even more than that, including the ability to enter values, sort columns, define alternating colors, etc.



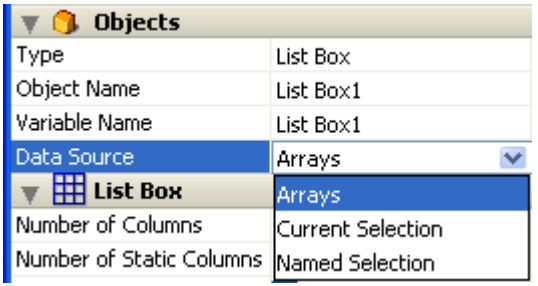
You can set up a List box completely in the 4D Form editor and can also manage it through programming. For more information on creating and setting List boxes in the Form editor as well as on their use, refer to the *Design Reference* manual of the 4D documentation. Programming List box objects is done in the same way as other 4D list form objects. However, specific rules must be followed, as detailed in this section.

Note: List box objects are designed for screen interfaces only. They cannot be printed.

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 SET FORMAT command.

Using the language, the values of columns (data entry and display) are managed using high-level List box commands (such as INSERT LISTBOX ROW or DELETE LISTBOX ROW) 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)
```

Note: When a List box object contains several columns, each related array must have the same size (same number of items) as the others, otherwise only the number of items of the smallest array will be displayed.

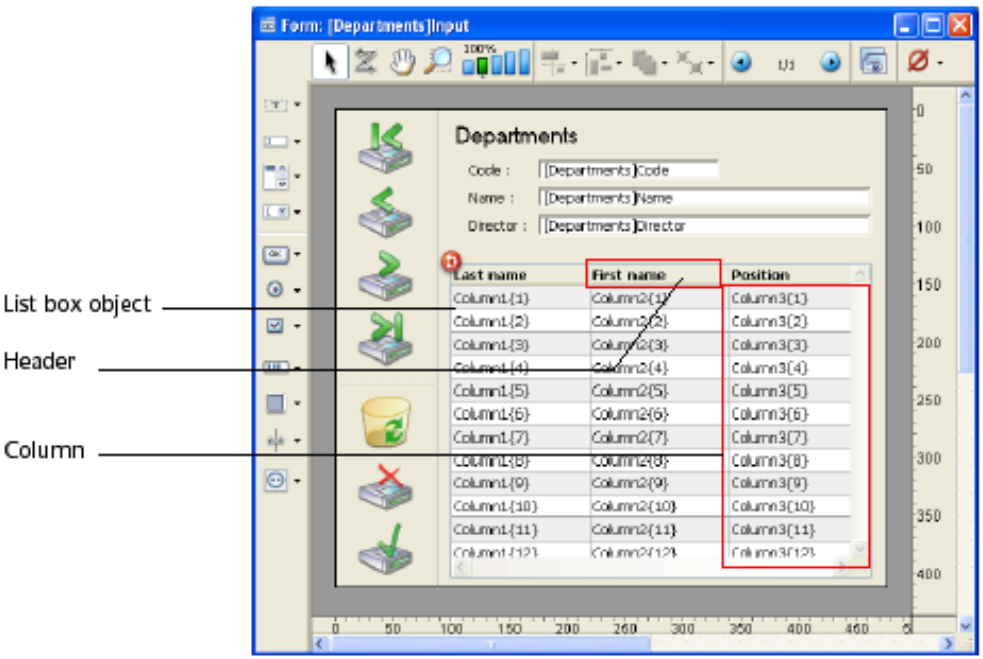
Warning : When a list box contains several columns of different sizes, only the number of items of the smallest array (column) will be displayed. You should make sure that each array has the same number of elements as the others. Also, if a list box column is empty (this occurs when the associated array 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 INSERT LISTBOX ROW and DELETE LISTBOX ROW 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 SET LISTBOX TABLE SOURCE command can be used to modify the table associated with the list box by programming.

Object, column and header

A List box object is composed of three separate items:
• the **object** itself,
• the **columns**,
• and the column **headers**.
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 X and headers are named Header1 to X in the form, independently of the list box objects.

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 and headers. 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 and/or column headers. To set the type of item on which you want to work, simply pass the 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
MOVE OBJECT	X		
GET OBJECT RECT	X		
SET FILTER		X	
SET FORMAT		X	
SET ENTERABLE		X	
SET CHOICE LIST		X	
BUTTON TEXT			X
SET COLOR	X	X	
SET RGB COLORS	X	X	
FONT	X	X	X
FONT SIZE	X	X	X
FONT STYLE	X	X	X
SET ALIGNMENT	X	X	X
Get alignment	X	X	X
SET VISIBLE	X	X	X
SET SCROLLBAR VISIBLE	X		
BEST OBJECT SIZE	X	X	X

Notes:
• All the commands of the "List Box" theme apply only to List box objects, except for the SET LISTBOX COLUMN WIDTH command (applies to object, column and header) and Get listbox column width command (applies to column and header only).
• Only the color of the text of column headers can be modified using the SET COLOR and SET RGB COLORS commands.

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.

SET VISIBLE and headers

When the SET VISIBLE command is used with a header, it is used on all List box object headers, regardless of the header set in the command. For example, the SET VISIBLE(*;"header3";False) instruction will hide all headers in the List box object to which header3 belongs and not simply this header.

Self, On Clicked and On Header Click

The Self function ("Language" theme) can be used in the object method of a list box or a list box column. In the case of a On Clicked or On Header Click form event, it returns respectively a pointer to the column variable or the header variable depending on where the click occurred.

Focus object

With an array type list box, the Focus object 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:=Focus object
    ` $Column contains a pointer to col2
$Row:= $Column->  `$Row equals 5
```

For a selection type list box, the Focus object 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.

SCROLL LINES

The SCROLL LINES command ("User Interface" 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.

Displayed line number

The Displayed line number command ("Selections" 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 MOVED LISTBOX ROW NUMBER and MOVED LISTBOX COLUMN NUMBER commands.

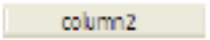
Managing sorts

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.

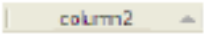
The developer can set up custom sorts using the SORT LISTBOX COLUMNS command and/or combining the On Header Click and On After Sort form events (see the Form event command) and array management 4D commands.

Note: The "Sortable" column property only affects the standard user sorts; the SORT LISTBOX COLUMNS command does not take this property into account.

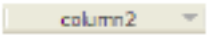
The value of the variable related to the column header allows you to manage additional information: the current sort of the column (read) and the display of the sort arrow.
• If the variable is set to 0, the column is not sorted and the sort arrow is not displayed;



• 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.

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 called "Highlight Set." 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 SELECT LISTBOX 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
```

DELETE LISTBOX COLUMN ({*; }object; colPosition{; number})

Parameter	Type		Description
*		→	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	Number	→	Column number to remove
number	Number	→	Number of columns to be removed

Description

The DELETE LISTBOX COLUMN command removes one or more columns (visible or invisible) in the list box set in the object et * 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.

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.

See Also

Get number of listbox columns, INSERT LISTBOX COLUMN.

DELETE LISTBOX ROW

version 2004

DELETE LISTBOX ROW ({*; }object; position)

Parameter	Type		Description
*		→	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)
position	Longint	→	Position of the row to delete

Description

The DELETE LISTBOX ROW command deletes the row number position (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 do not pass a string, but a variable reference. 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 position row is automatically removed from all the arrays used by the list box columns. If the position value is higher than the total number of rows in the list box, the command does nothing.

See Also

Get number of listbox rows, INSERT LISTBOX ROW.

GET LISTBOX ARRAYS ({*; }object; arrColNames; arrHeaderNames; arrColVars; arrHeaderVars; arrVisible; arrStyles)

Parameter	Type		Description
*		→	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	Array string	←	Column object names
arrHeaderNames	Array string	←	Header object names
arrColVars	Array pointer	←	Pointers to column variables or Pointers to column fields or Nil
arrHeaderVars	Array pointer	←	Pointers to header variables
arrVisible	Array Boolean	←	Visibility of each column
arrStyles	Array pointer	←	Pointers to arrays or style and color variables or Nil

Description

The GET LISTBOX 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 do not pass a string, but a variable reference. 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 arrVisible 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, three pointers to three arrays that allow the applying of a specific style, font color and background color to each row of the list box. These arrays are associated with the list box in the Property List of the Design environment. If an array is not specified for the list box, the corresponding item in arrStyles will contain a Nil pointer.
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.

See Also

Get listbox information.

GET LISTBOX CELL POSITION ({*; }object; column; row{; colVar})

Parameter	Type		Description
*		→	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 GET LISTBOX 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.

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, GET LISTBOX 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.

The values returned by the command are not updated in the case of a right mouse click (or Control+click under Mac OS) on the header of a list box column..

Get listbox column width

Get listbox column width ({*; }object) → Integer

Parameter	Type		Description
*		→	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	Integer	←	Column width (in pixels)

Description

The Get listbox 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 do not pass a string, but a variable reference. For more information about object names, refer to the Object Properties section.

See Also

SET LISTBOX COLUMN WIDTH.

Get listbox information

version 2004.1 (Modified)

Get listbox information ({*; }object; info) → Longint

Parameter	Type		Description
*		→	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 Get listbox information command returns various information regarding the current visibility and size of headers 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 do not pass a string, but a variable reference. For more information about object names, refer to the Object Properties section.

In the dans 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	Returned value(s)
Display listbox header	Longint	0	0=hidden, 1=shown
Listbox header height	Longint	1	Height in pixels
Display listbox hor scrollbar	Longint	2	0=hidden, 1=shown
Listbox hor scrollbar height	Longint	3	Height in pixels
Display listbox ver scrollbar	Longint	4	0=hidden, 1=shown
Listbox ver scrollbar width	Longint	5	Width in pixels
Position listbox hor scrollbar	Longint	6	Position of the cursor in pixels
Position listbox ver scrollbar	Longint	7	Position of the cursor in pixels

• The first six constants are useful for calculating the actual size of a list box area in a form.
• When you use the constants Position listbox hor scrollbar or Position listbox ver scrollbar, the Get listbox 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.

Example

Given a list box containing rows with a height of 20 pixels each. You execute the following statement:

```
$scroll:=Get listbox information(*;"Listbox";Position listbox ver scrollbar)
```

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).

See Also

SET SCROLLBAR VISIBLE, SHOW LISTBOX GRID.

Get listbox rows height

version 2004

Get listbox rows height ({*; }object) → Integer

Parameter	Type		Description
*		→	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	Integer	←	Row height (in pixels)

Description

The Get listbox rows height command returns the current row height (in pixels) 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 do not pass a string, but a variable reference. For more information about object names, refer to the Object Properties section.

See Also

SET LISTBOX ROWS HEIGHT.

GET LISTBOX TABLE SOURCE ({*; }object; tableNum{; name})

Parameter	Type		Description
*	*	→	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

Description

The GET LISTBOX TABLE SOURCE command can be used to find out the current source of the data displayed in the listbox 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 do not pass a string but a variable reference. 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 listbox 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.

See Also

SET LISTBOX TABLE SOURCE.

Get number of listbox columns

version 2004

Get number of listbox columns ({*; }object) → Longint

Parameter	Type		Description
*		→	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 Get number of listbox 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 do not pass a string, but a variable reference. For more information on object names, refer to the Object Properties section.

See Also

DELETE LISTBOX COLUMN.

Get number of listbox rows

version 2004

Get number of listbox rows ({*; }object) → Longint

Parameter	Type		Description
*		→	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 Get number of listbox rows command returns the number of rows 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 do not pass a string, but a variable reference. 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.

See Also

DELETE LISTBOX ROW, INSERT LISTBOX ROW.

INSERT LISTBOX COLUMN ({*; }object; colPosition; colName; colVariable; headerName; headerVar)

Parameter	Type		Description
*		→	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	Number	→	Location of column to insert
colName	String	→	Name of the column object
colVariable	Array | Field | Var	→	Column array name or field or variable
headerName	String	→	Name of the column header object
headerVar	Integer variable	→	Column header variable

Description

The INSERT LISTBOX COLUMN command inserts a column in the list box set 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 do not pass a string, but a variable reference. 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.
• 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 Management of 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, INSERT LISTBOX 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 INSERT LISTBOX 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.

Note: Object names must be unique in a form. You must be sure that the names passed in the colName and headerName parameters are not already used. Otherwise, the column is not created and an error is generated.

Examples

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)

$Last:=Get number of listbox columns(*;"ListBox1")+1
INSERT LISTBOX COLUMN(*;"ListBox1";$Last;"ColumnPicture";Picture;"HeaderPicture";
                                                        HeaderVarName)
```

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:

```
$last:=Get number of listbox columns(*;"ListBox1")+1
INSERT LISTBOX COLUMN(*;"ListBox1";$last;"FieldCol";[Transport]Fees;"HeaderName";
                                                        HeaderVar)
```

See Also

DELETE LISTBOX COLUMN, INSERT LISTBOX COLUMN FORMULA.

INSERT LISTBOX COLUMN FORMULA

version 11

INSERT LISTBOX COLUMN FORMULA ({*; }object; colPosition; colName; formula; dataType; headerName; headerVariable)

Parameter	Type		Description
*	*	→	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	Number	→	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
headerVariable	Integer variable	→	Column header variable

Description

The INSERT LISTBOX COLUMN FORMULA command inserts a column into the listbox designated by the object and * parameters.
The INSERT LISTBOX COLUMN FORMULA command is similar to the INSERT LISTBOX 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 Management of List box objects section).

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, 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	Valeur
Is Real	Longint	1
Is Text	Longint	2
Is Picture	Longint	3
Is Date	Longint	4
Is Boolean	Longint	6
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 headerVariable parameters, pass the object name and variable of the column header inserted.

Note: Object names must be unique in a form. You need to make sure that the names passed in the colName and headerName parameters are not already used. Otherwise, the column is not created and an error is generated.

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"
$last:=Get number of listbox columns(*;"ListBox1")+1
INSERT LISTBOX COLUMN FORMULA(*;"ListBox1";$last;"ColFormula";Is Real;vAge;
                                                              "Age";HeaderVar)
```

See Also

INSERT LISTBOX COLUMN.

INSERT LISTBOX ROW

version 2004

INSERT LISTBOX ROW ({*; }object; position)

Parameter	Type		Description
*		→	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)
position	Longint	→	Position of the row to insert

Description

The INSERT LISTBOX ROW command inserts a new row 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 do not pass a string, but a variable reference. For more information about object names, refer to the Object Properties section.

The row is inserted at the position set by the position parameter. A new row is automatically added at this position in all the arrays used by the list box columns, whatever their type and their visibility.

If the position value is higher than the total number of rows in the list box, the row is added at the end of each array. If it is equal to 0, the row is added at the beginning of each array. If it contains a negative value, the command does nothing.

See Also

DELETE LISTBOX ROW.

MOVED LISTBOX COLUMN NUMBER ({*; }object; oldPosition; newPosition)

Parameter	Type		Description
*		→	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	Number	←	Previous position of the moved column
newPosition	Number	←	New position of the moved column

Description

The MOVED LISTBOX 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 do not pass a string, but a variable reference. 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.

See Also

Form event, MOVED LISTBOX ROW NUMBER.

MOVED LISTBOX ROW NUMBER

version 2004

MOVED LISTBOX ROW NUMBER ({*; }object; oldPosition; newPosition)

Parameter	Type		Description
*		→	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	Number	←	Previous position of the moved row
newPosition	Number	←	New position of the moved row

Description

The MOVED LISTBOX ROW NUMBER command returns two numbers in oldPosition and newPosition indicating respectively the previous position and the new position of the row moved in 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 do not pass a string, but a variable reference. 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).

See Also

Form event, MOVED LISTBOX COLUMN NUMBER.

SELECT LISTBOX ROW ({*; }object; position{; action})

Parameter	Type		Description
*		→	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)
position	Longint	→	Number of the row to select
action	Longint	→	Selection action

Description

The SELECT LISTBOX 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 do not pass a string, but a variable reference. 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):
• Replace listbox selection (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. This is the default action (if the action parameter is omitted).
• Add to listbox selection (1): The row selected is added to the existing selection. If the row specified already belongs to the existing selection, the command does nothing.
• Remove from listbox selection (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.

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 Replace listbox selection or is omitted, all the rows of the listbox are selected. If the action parameter contains Remove from listbox selection, all the listbox rows are deselected.

• If the position value is greater than the total number of rows contained in the 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 SELECT LISTBOX 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 SCROLL LINES command.
• To switch a row into editing mode (to allow data entry), use the EDIT ITEM command.

See Also

DELETE LISTBOX ROW, EDIT ITEM, INSERT LISTBOX ROW, SCROLL LINES.

SET LISTBOX COLUMN WIDTH ({*; }object; width)

Parameter	Type		Description
*		→	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	Integer	→	Column width (in pixels)

Description

The SET LISTBOX 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 paraemter, 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.

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.

See Also

Get listbox column width.

SET LISTBOX GRID COLOR

version 2004

SET LISTBOX GRID COLOR ({*; }object; color; horizontal; vertical)

Parameter	Type		Description
*		→	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	Number	→	RGB color value
horizontal	Boolean	→	Use color for horizontal grid lines
vertical	Boolean	→	Use color for vertical grid lines

Description

The SET LISTBOX 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 do not pass a string, but a variable reference. 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 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.

See Also

SET RGB COLORS, SHOW LISTBOX GRID.

SET LISTBOX ROWS HEIGHT

version 2004

SET LISTBOX ROWS HEIGHT ({*; }object; height)

Parameter	Type		Description
*		→	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	Integer	→	Row height (in pixels)

Description

The SET LISTBOX 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 do not pass a string, but a variable reference. For more information about object names, refer to the Object Properties section.

See Also

Get listbox rows height.

SET LISTBOX TABLE SOURCE

version 11

SET LISTBOX TABLE SOURCE ({*; }object; tableNum | name)

Parameter	Type		Description
*	*	→	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 | name	Longint |String	→	Number of table whose current selection is to be used or Named selection to be used

Description

The SET LISTBOX 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 Management of 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 do not pass a string but a variable reference. 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 name parameter, the listbox will be filled in with the data of the records belonging to the named selection.

If the listbox already contains columns, their contents will be updated after the command is executed.

See Also

GET LISTBOX TABLE SOURCE.

SHOW LISTBOX GRID ({*; }object; horizontal; vertical)

Parameter	Type		Description
*		→	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 SHOW LISTBOX 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 do not pass a string, but a variable reference. 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 valeurs (True) or hidden (False). The grid is displayed by default.

See Also

Get listbox information, SET LISTBOX GRID COLOR.

SORT LISTBOX COLUMNS ({*; }object; colNum; order{; colNum2; order2; ...; colNumN; orderN})

Parameter	Type		Description
*		→	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	> or <	→	> to sort in ascending order or < to sort in descending order

Description

The SORT LISTBOX 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 do not pass a string, but a variable reference. 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.

26

Math

Abs (number) → Number

Parameter	Type		Description
number	Number	→	Number for which to return the absolute value
Function result	Number	←	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 (number) → Number

Parameter	Type		Description
number	Number	→	Tangent for which to calculate the angle
Function result	Number	←	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...).

Examples

The following example shows the value of Pi:

```
ALERT("Pi is equal to: "+String(Arctan(1)*4))
```

See Also

Cos, Sin, Tan.

Cos (number) → Number

Parameter	Type		Description
number	Number	→	Number, in radians, whose cosine is returned
Function result	Number	←	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...).

See Also

Arctan, Sin, Tan.

Dec (number) → Number

Parameter	Type		Description
number	Number	→	Number whose decimal portion is returned
Function result	Number	←	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
```

See Also

Int.

Preliminary Note
If you do not deal with cross-platform development, you can skip this section.

On computers, floating point arithmetic is more a technology than a mathematical science. For example, you learned in school that one-third (1/3) can be written as an infinite number of threes after the decimal point. A computer, on the other hand, does not know this and must calculate the expression. In the same way, you know conceptually that three times one third is equal to one; a computer calculates the expression to get the result. Depending on the type of computer you use, one-third is calculated as a limited number of threes after the decimal point. This number is called the "precision" of the machine.

On older 68K-based Macintosh, the precision number is 19; this means that 1/3 is calculated with 19 significant digits. On Windows and Power Macintosh, this number is 15; so 1/3 is displayed with 15 significant digits. If you display the expression 1/3 in the Debugger window of 4D, you will get 0.3333333333333333333 on 68K-based Macintosh and something like 0.333333333333333148 on Windows or Power Macintosh. Note that the last three digits are different because the precision on Windows and Power Macintosh is less than on the 68K-based Macintosh. Yet, if you display the expression (1/3)*3, the result is 1 on both machines.

If your floating point arithmetic computations deal with the number of square feet in your backyard, you will say "Fine with me!" because you do not care about the digits after the decimal point. On the other hand, if you are filling out an IRS form, you may, in certain circumstances, care about the accuracy of your computer. However, remember that 19 or 15 digits after the decimal point are quite sufficient even if you manage billions of dollars of revenue.

Why does the value 1/3 seem different on 68K Macintosh and onWindows or Power Macintosh?

On 68K-based Macintosh, the operating system stores real numbers on 10 bytes (80 bits), while on Windows and Power Macintosh, it stores them on 8 bytes (64 bits). This is why real numbers have up to 19 significant digits on 68K-based Macintosh and up to 15 significant digits on Windows and Power Macintosh.

So, why does the expression (1/3)*3 return 1 on both machines?

A computer can only make approximate computations. Therefore, while comparing or computing numbers, a computer does not treat real numbers as mathematical objects but as approximate values. In our example, 0.3333... multiplied by 3 gives 0.9999...; the difference between 0.9999... and 1 is so small that the machine considers the result equal to 1, and consequently returns 1. For details on this subject, see the discussion for the command SET REAL COMPARISON LEVEL.

There is dual behavior of real numbers, so we must make the distinction between:
• How they are calculated and compared
• How they are displayed on the screen or printer
Originally, 4D handled real numbers using the standard 10-byte data type provided by the operating system of the 68K-based Macintosh. Consequently, real values stored in the data file on disk are saved using this format. In order to maintain compatibility between the 68K, Power Macintosh, and Windows versions of 4D, the 4D data files still hold the real values using the 10-byte data type. Because floating point arithmetic is performed on Windows or Power Macintosh using the 8 byte format, 4D converts the values from 10 bytes to 8 bytes, and vice versa. Therefore, if you load a record containing real values, which have been saved on 68K-based Macintosh, onto Windows or Power Macintosh, it is possible to lose some precision (from 19 to 15 significant digits). Yet, if you load a record containing real values, which have been saved on Windows or Power Macintosh, on a 68K-based Macintosh, there will be no loss of precision. Basically, if you use a database on 68K or Power Macintosh and Windows, count on floating point arithmetic with 15 significant digits, not 19.

Using the SET DATABASE PARAMETER command, you can set the number of digits to be skipped (4 by default) when simplifying the display of real numbers.

Euro converter (value; fromCurrency; toCurrency) → Real

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 allows you to convert any value from and to the different currencies belonging to the “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
Deutschemark	String	DEM
Euro	String	EUR
Finnish Markka	String	FIM
French Franc	String	FRF

Greek drachma	String	GRD
Irish Pound	String	IEP
Italian Lire	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 (number) → Number

Parameter	Type		Description
number	Number	→	Number to evaluate
Function result	Number	←	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...
```

See Also

Log.

Int (number) → Number

Parameter	Type		Description
number	Number	→	Number whose integer portion is returned
Function result	Number	←	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
```

See Also

Dec.

Log (number) → Number

Parameter	Type		Description
number	Number	→	Number for which to return the log
Function result	Number	←	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))))
```

See Also

Exp.

Mod (number1; number2) → Number

Parameter	Type		Description
number1	Number	→	Number to divide
number2	Number	→	Number to divide by
Function result	Number	←	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
```

See Also

Numeric Operators.

Random → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Random number

Description

Random returns a random integer value between 0 and 32,767 (inclusive).

To define a range of integers, use this formula:

```
(Random%(End–Start+1))+Start
```

The value start is the first number in the range, and the value end 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; places) → Number

Parameter	Type		Description
round	Number	→	Number to be rounded
places	Number	→	Number of decimal places used for rounding
Function result	Number	←	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
```

See Also

Trunc.

SET REAL COMPARISON LEVEL

version 6.0

SET REAL COMPARISON LEVEL (epsilon)

Parameter	Type		Description
epsilon	Number	→	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.

Note: If you want to execute a query or an "Order by" on a numeric indexed field whose values are lower than 10^-6, make sure that the SET REAL COMPARISON LEVEL command is executed before construction of the index.

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.

See Also

Comparison Operators.

Sin (number) → Number

Parameter	Type		Description
number	Number	→	Number, in radians, whose sine is returned
Function result	Number	←	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...).

See Also

Arctan, Cos, Tan.

Square root (number) → Number

Parameter	Type		Description
number	Number	→	Number whose square root is calculated
Function result	Number	←	Square root of the number

Description

Square root returns the square root of number.

Examples

1. The line:

```
$vrSquareRootOfTwo := Square root (2)
```

assigns the value 1.414213562373 to the variable $vrSquareRootOfTwo.

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.

See Also

Numeric Operators.

Tan (number) → Number

Parameter	Type		Description
number	Number	→	Number, in radians, whose tangent is returned
Function result	Number	←	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...).

See Also

Arctan, Cos, Sin.

Trunc (number; places) → Number

Parameter	Type		Description
number	Number	→	Number to be truncated
places	Number	→	Number of decimal places used for truncating
Function result	Number	←	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
```

See Also

Round.

27

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 acccess 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, thanks to 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 (menu; itemText{; subMenu{; process}})

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
itemText	Text	→	Text for the new menu items
subMenu	MenuRef	→	Reference of submenu associated with item
process	Number	→	Process reference number

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 purpose and will be ignored.

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 "(-" 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.

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)
```

See Also

DELETE MENU ITEM, INSERT MENU ITEM, SET MENU ITEM REFERENCE.

Count menu items

version 11 (Modified)

Count menu items (menu{; process}) → Number

Parameter	Type		Description
menu	Num | MenuRef	→	Menu number or Menu reference
process	Number	→	Process reference number
Function result	Number	←	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.

See Also

Count menus.

Count menus

version 6.0

Count menus {(process)} → Number

Parameter	Type		Description
process	Number	→	Process reference number
Function result	Number	←	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.

See Also

Count menu items.

Create menu {(menu)} → MenuRef

Parameter	Type		Description
menu	MenuRef | Num | Alpha	→	Menu reference or Number or Name of menu bar
Function result	MenuRef	←	Menu reference

Description

The Create menu command can be used to create 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 APPEND MENU ITEM, 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 souce 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.

A menu created by this command can be used as the menu bar using the SET MENU BAR command.

Example

Refer to the example of the SET MENU BAR command.

See Also

Dynamic pop up menu, RELEASE MENU, SET MENU BAR.

DELETE MENU ITEM

version 11 (Modified)

DELETE MENU ITEM (menu; menuItem{; process})

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for last item added
process	Number	→	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.

See Also

APPEND MENU ITEM, INSERT MENU ITEM.

DISABLE MENU ITEM (menu; menuItem{; process})

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for the last item added
process	Number	→	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.

See Also

ENABLE MENU ITEM.

Dynamic pop up menu

version 11

Dynamic pop up menu (menu{; default{; xCoord{; yCoord}}}) → ItemRef

Parameter	Type		Description
menu	MenuRef	→	Menu reference
default	ItemRef	→	Reference of item selected by default
xCoord	Number	→	X coordinate of top left corner
yCoord	Number	→	Y coordinate of top left corner
Function result	ItemRef	←	Reference of selected menu item

Description

The Dynamic pop up menu causes a hierarchical pop-up menu to appear at the current location of the mouse or at the location set by the optional xCoord and yCoord parameters. The hierarchical menu used must have been created using the Create menu command. The reference returned by Create menu must then be passed in the menu parameter.

In conformity with standard interface rules, this command must generally be called in response to a right mouse click or when the button is held down a certain period of time (contextual menu for example).

The optional default parameter can be used to set an item of the popup menu as selected by default whenever the menu appears. In this parameter, pass a menu item reference. This reference must have been set beforehand using the SET MENU ITEM REFERENCE command. If you do not pass this parameter, the first item of the menu will be selected by default.

The optional xCoord and yCoord parameters can be used to specify the location of the pop-up menu to be displayed. In the xCoord and yCoord parameters, pass the horizontal and vertical coordinates, respectively, of the top left corner of the menu. These coordinates must be expressed in pixels in the local coordinate system of the current form. Both parameters must be passed together; if only one of them is passed, it will be ignored.

If you want to display a pop-up menu associated with a 3D button, then do not pass the optional xCoord and yCoord parameters. In this case, 4D automatically calculates the location of the menu with respect to the button according to the interface standards of the current platform.

If a menu item has been selected, the command returns its reference (such as it has been defined using the SET MENU ITEM REFERENCE command). Otherwise, the command returns an empty string.

Note: The existing Pop up menu command ("User Interface" theme) can be used to create pop-up menus based on text.

See Also

Get menu item reference, Get selected menu item reference, Pop up menu, SET MENU ITEM REFERENCE.

ENABLE MENU ITEM (menu; menuItem{; process})

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for the last item added
process	Number	→	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.

See Also

DISABLE MENU ITEM.

Get menu bar reference {(process)} → MenuRef

Parameter	Type		Description
process	Number	→	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.

See Also

SET MENU BAR.

Get menu item

version 11 (Modified)

Get menu item (menu; menuItem{; process}) → String

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for last item added
process	Number	→	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.

See Also

Get menu item key, SET MENU ITEM.

GET MENU ITEM ICON (menu; menuItem; iconRef{; process})

Parameter	Type		Description
menu	MenuRef | Num	→	Menu reference or Menu number
menuItem	Number	→	Number of menu item or -1 for the last item added to the menu
iconRef	Text var | Longint var	←	Name or number of library picture of icon associated with menu item
process	Number	→	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 in the picture library.

The icon associated with a menu item is added to the tool bar of the application.

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 iconRef, the command returns either the name or number of the picture depending on the type of variable passed in this parameter. 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 picture.

See Also

SET MENU ITEM ICON.

Get menu item key

version 11 (Modified)

Get menu item key (menu; menuItem{; process}) → Number

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for the last item added
process	Number	→	Process reference number
Function result	Number	←	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
```

See Also

Get menu item key, SET MENU ITEM SHORTCUT.

Get menu item mark

version 11 (Modified)

Get menu item mark (menu; menuItem{; process}) → String

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for last item added
process	Number	→	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 command SET MENU ITEM MARK.

Example

The following example toggles the check mark of a menu item:

```
SET MENU ITEM MARK($vlMenu;$vlItem;Char(18)*Num(Get menu item mark($vlMenu;
                                                $vlItem)=""))
```

See Also

SET MENU ITEM MARK.

Get menu item method

Get menu item method (menu; menuItem{; process}) → String

Parameter	Type		Description
menu	MenuRef | Num	→	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.

See Also

SET MENU ITEM METHOD.

Get menu item modifiers

version 11

Get menu item modifiers (menu; menuItem{; process}) → Number

Parameter	Type		Description
menu	MenuRef | Longint	→	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	Number	←	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 code of 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.

See Also

Get menu item key, SET MENU ITEM SHORTCUT.

GET MENU ITEM PROPERTY

version 11

GET MENU ITEM PROPERTY (menu; menuItem; property; value{; process})

Parameter	Type		Description
menu	MenuRef | Longint	→	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.

See Also

SET MENU ITEM PROPERTY.

Get menu item reference

version 11

Get menu item reference (menu; menuItem) → ItemRef

Parameter	Type		Description
menu	MenuRef | Longint	→	Menu reference or Menu number
menuItem	Longint	→	Number of menu item or -1 for the last item added to the menu
Function result	ItemRef	←	Reference of the menu item

Description

La command Get menu item reference retourne la See Also de la ligne de menu désignée par les Parameters menu et ligneMenu. Cette See Also doit avoir été préalablement définie à l'aide de la command FIXER See Also LIGNE MENU.
The Get menu item reference command returns the reference of the menu item designated by the menu and menuItem parameters. This reference must have been set beforehand using the SET MENU ITEM REFERENCE command.

See Also

Dynamic pop up menu, Get selected menu item reference, SET MENU ITEM REFERENCE.

Get menu item style

version 11 (Modified)

Get menu item style (menu; menuItem; itemStyle{; process})

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for last item added
itemStyle	Number	→	New menu item style
process	Number	→	Process reference number

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
Plain	Long Integer	0
Bold	Long Integer	1
Italic	Long Integer	2
Underline	Long Integer	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
```

See Also

SET MENU ITEM STYLE.

GET MENU ITEMS (menu; menuTitlesArray; menuRefsArray)

Parameter	Type		Description
menu	MenuRef | Longint	→	Menu reference or Menu number
menuTitlesArray	Alpha array (32)	←	Array of menu titles
menuRefsArray	Alpha array (16)	←	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 STRING(32;menuTitlesArray;0)
ARRAY STRING(16;menuRefsArray;0)
MenuBarRef:=Get menu bar reference(Frontmost process)
GET MENU ITEMS(MenuBarRef;menuTitlesArray;menuRefsArray)
```

Get menu title

version 11 (Modified)

Get menu title (menu{; process}) → String

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
process	Number	→	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, Get menu title 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.

See Also

Count menus.

Get selected menu item reference

version 11

Get selected menu item reference → ItemRef

Parameter	Type		Description
This command does not require any parameters			
Function result	ItemRef	←	Reference of the menu item

Description

The Get selected menu item reference command returns the reference of the selected menu item. This reference must have been set beforehand using the SET MENU ITEM REFERENCE command.
If no menu item has been selected, the command returns an empty string "".

See Also

Dynamic pop up menu, Get menu item reference, SET MENU ITEM REFERENCE.

INSERT MENU ITEM (menu; afterItem; itemText{; subMenu{; process}})

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
afterItem	Number	→	Menu item number
itemText	String	→	Text for the menu item to be inserted
subMenu	MenuRef	→	Reference of submenu associated with item
process	Number	→	Process reference number

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, INSERT 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.

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 the item definition passed in itemText.

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")
```

See Also

APPEND MENU ITEM, SET MENU ITEM REFERENCE.

Menu selected {(subMenu)} → Number

Parameter	Type		Description
subMenu	MenuRef	←	Reference of menu containing item selected
Function result	Number	←	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. However, if you use the commands APPEND MENU ITEM or INSERT MENU ITEM, you have to use Menu selected because the menu items added by these commands do not have associated methods.

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.

See Also

Managing Menus.

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 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 will not be removed. If you want to remove a menu and its sub-menus, you must remove each of the submenus individually.

See Also

Create menu.

SET MENU BAR (menuBar{; process{; *}})

Parameter	Type		Description
menuBar	Number| String | MenuRef	→	Number or name of the menu bar or Menu reference
process	Number	→	Process reference number
*		→	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.

Examples

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)
```

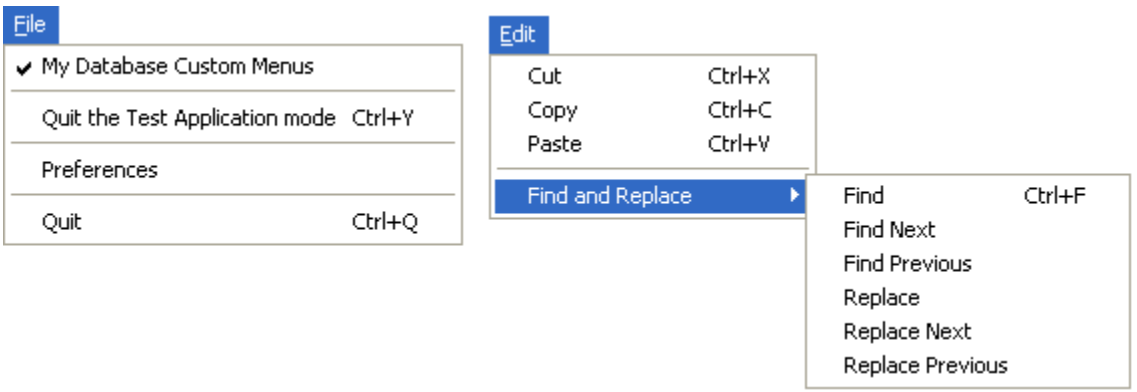
2. The following example changes the current menu bar to the menu bar named "FormMenuBar1" and saves the states of the menu commands. Menu commands that were previously disabled will appear disabled.

```
SET MENU BAR ("FormMenuBar1";*)
```

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;*)
```

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_STRING(16;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))
      `Preferences
SET MENU ITEM PROPERTY(FileMenu;6;Associated Standard Action; Preferences Action)
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 File and Replace menu
       `FindAndReplaceMenu will contain the Find and Replace menu reference
C_STRING(16;FindAndReplaceMenu)
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_STRING(16; 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;"(-")
       `item that will have submenu
INSERT MENU ITEM(EditMenu;-1;"Find and Replace" ; FindAndReplaceMenu)
```

```
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)
```

See Also

Managing Menus.

SET MENU ITEM (menu; menuItem; itemText{; process})

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for the last item added
itemText	String	→	New text for the menu item
process	Number	→	Process reference number

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 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.

See Also

Get menu item, SET MENU ITEM SHORTCUT.

SET MENU ITEM ICON (menu; menuItem; iconRef{; process})

Parameter	Type		Description
menu	MenuRef | Num	→	Menu reference or Menu number
menuItem	Number	→	Number of menu item or -1 for the last item added to the menu
iconRef	Text | Longint	→	Name or number of library picture to be associated with menu item
process	Number	→	Process number

Description

The SET MENU ITEM ICON command can be used to modify 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.

The icon associated with a menu item is added to the tool bar of the application. The picture will be displayed in a 20 x 20 pixel frame.
In iconRef, you can pass either the name or number of the library picture to be used as the icon. 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.

See Also

GET MENU ITEM ICON.

version 11 (Modified)

SET MENU ITEM MARK (menu; menuItem; mark{; process})

Parameter	Type		Description
menu	Number | RefMenu	→	Menu number or Menu reference
menuItem	Number	→	Item number or -1 for last item added
mark	String	→	New menu item mark
process	Number	→	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, SET 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 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.

See Also

Get menu item mark.

SET MENU ITEM METHOD (menu; menuItem; methodName{; process})

Parameter	Type		Description
menu	MenuRef | Longint	→	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 can be used to modify 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.

See Also

Get menu item method.

SET MENU ITEM PROPERTY (menu; menuItem; property; value{; process})

Parameter	Type		Description
menu	MenuRef | Longint	→	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 is used to set the value of the property for 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 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:

property (Constant)	value (Possible values)
Associated Standard Action Used to associate a standard action with a menu item.	0 = No Action 1 = Cancel Action 2 = Accept Action 3 = Next record Action 4 = Previous record Action 5 = First record Action 6 = Last record Action 7 = Delete record Action

	8 = Next page Action 9 = Previous page Action 10 = First page Action 11 = Last page Action 12 = Edit subrecord Action 13 = Delete subrecord Action 14 = Add subrecord Action 17 = Undo Action 18 = Cut Action 19 = Copy Action 20 = Paste Action 21 = Clear Action 22 = Select all Action 23 = Show Clipboard Action 26 = Test Application Action 27 = Quit Action 31 = Redo Action 32 = Preferences Action 35 = Return to Design mode 36 = MSC Action
Start a New Process Used to activate the "Start New Process" option.	0 = Yes 1 = No
Access Privileges Used to assign an access group to the command.	0 = All Groups >0 = Group ID

For more information about standard menu item properties, refer to the “Creating Custom Menus” chapter 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.

See Also

GET MENU ITEM PROPERTY.

SET MENU ITEM REFERENCE

version 11

SET MENU ITEM REFERENCE (menu; menuItem; itemRef)

Parameter	Type		Description
menu	MenuRef | Longint	→	Menu reference or Menu number
menuItem	Longint	→	Number of menu item or -1 for the last item added to the menu
itemRef	String	→	String to associate as reference

Description

The SET MENU ITEM REFERENCE command can be used to associate a custom reference with a menu item designated by the menu and menuItem parameters.

This reference will mainly be used by the Dynamic pop up menu command.

See Also

Dynamic pop up menu, Get menu item reference, Get selected menu item reference.

SET MENU ITEM SHORTCUT (menu; menuItem; itemKey{; modifiers{; process}})

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for last item added
itemKey	Number | Text	→	Character code of keyboard shortcut or Letter of keyboard shortcut
modifiers	Longint	→	Modifier(s) to associate with shortcut (ignored if key code is passed)
process	Number	→	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 will automatically be combined with the **Ctrl** (Windows) or **Command** (Macintosh) key in order to set the new keyboard shortcut.

You can pass the name of the key directly as text (a letter) in the itemKey parameter, for example "U" to specify the **Ctrl+U** (Windows) or **Command+U** (Mac OS) shortcut. When you use this syntax, you can also pass the optional modifiers parameter in order to associate additional modifiers with the shortcut. This way you can define shortcuts of the **Ctrl+Alt+Shift+Z** (Windows) or **Cmd+Option+Shift+Z** (Mac OS) type.
To do this, pass the following values in modifiers:
• 512 for the **Shift** key
• 2048 for the **Option** (Mac OS) or **Alt** (Windows) key
• To associate both keys, combine their values.
Note that the **Ctrl** (Windows) and **Command** (Mac OS) keys are automatically added by 4D to the keyboard shortcut.

Note: You can specify the value to pass using the Shift key mask and Option key mask constants of the "Events (Modifiers)" theme.

The modifiers parameter is not taken into account if it is passed when the modifier key is specified via its character code (former syntax).

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

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)
```

See Also

Get menu item key, Get menu item modifiers.

SET MENU ITEM STYLE (menu; menuItem; itemStyle{; process})

Parameter	Type		Description
menu	Number | MenuRef	→	Menu number or Menu reference
menuItem	Number	→	Menu item number or -1 for last item added
itemStyle	Number	→	New menu item style
process	Number	→	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
Plain	Long Integer	0
Bold	Long Integer	1
Italic	Long Integer	2
Underline	Long Integer	4

See Also

Get menu item style.

28

Messages

ALERT (message{; ok button title})

Parameter	Type		Description
message	String	→	Message to display in the alert dialog box
ok button title	String	→	OK button title

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 ok button title. If necessary, the OK button width is resized toward the left, according to the width of the custom title you pass.

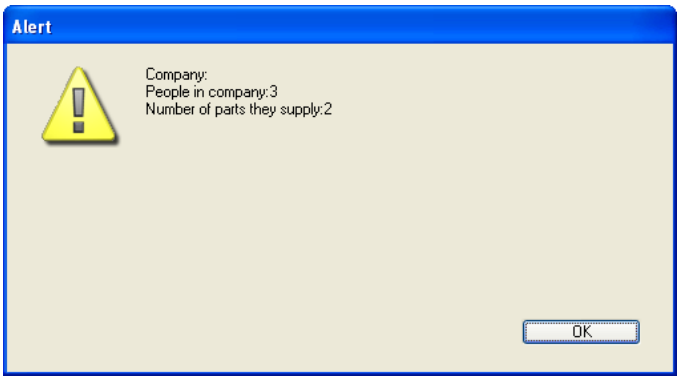
Tip: Do not call the ALERT command from the section of a form or object method that handles the On Activate or On Deactivate form events; this will cause an endless loop.

Examples

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])))
```

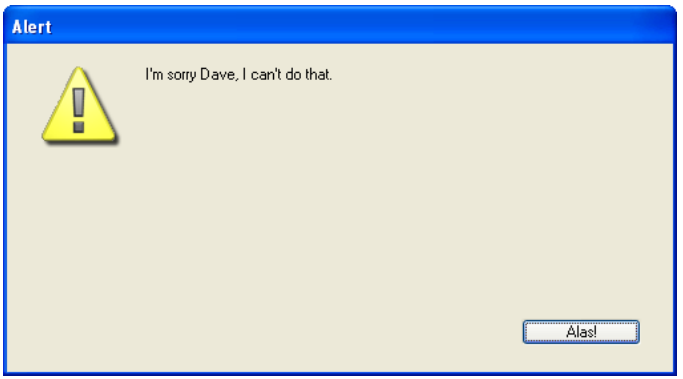
This line of code displays the following alert box (on Windows):



2. The line:

ALERT("I'm sorry Dave, I can't do that.";"Alas!")

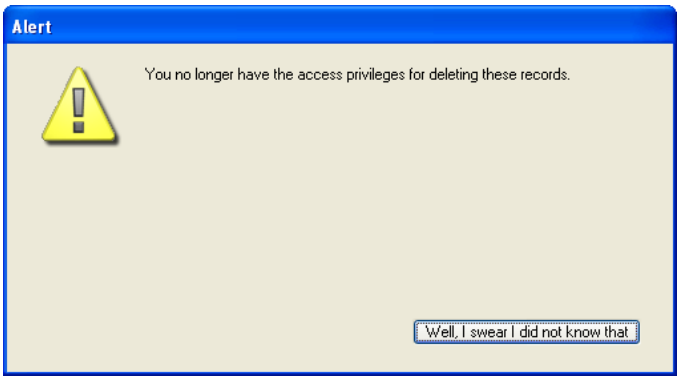
displays the alert dialog box (on Windows) shown:



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:



See Also

CONFIRM, DISPLAY NOTIFICATION, Request.

CONFIRM (message{; OK button title{; cancel button title}})

Parameter	Type		Description
message	String	→	Message to display in the confirmation dialog box
OK button title	String	→	OK button title
cancel button title	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 ok button title and cancel button title. 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.

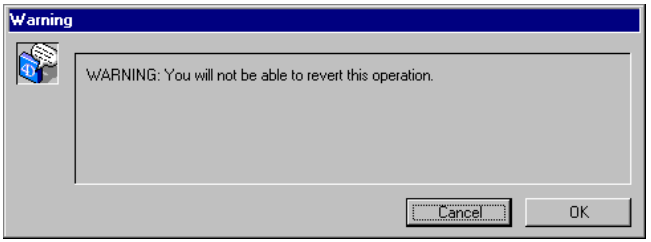
Tip: Do not call the CONFIRM command from the section of a form or object method that handles the On Activate or On Deactivate form events; this will cause an endless loop.

Examples

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
```

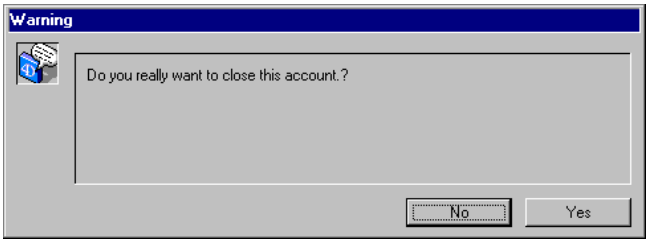
will display the confirm dialog box (on Windows) shown here:



2. The line:

```
CONFIRM("Do you really want to close this account?";"Yes";"No")
```

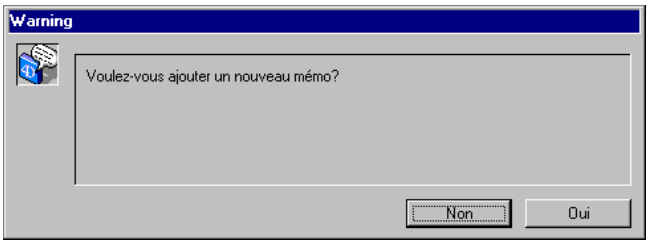
will display the confirm dialog box (on Windows) shown here:



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})
```

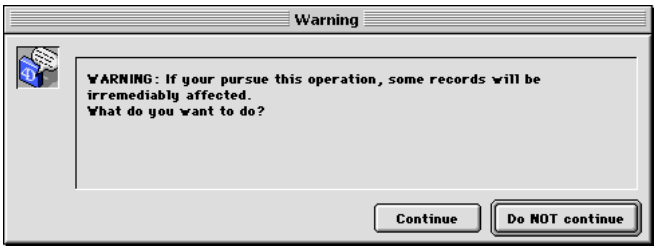
could display the French confirm dialog box (on Windows) shown here:



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:



See Also

ALERT, Request.

DISPLAY NOTIFICATION (title; text{; duration})

Parameter	Type		Description
title	Alpha 255	→	Notification title
text	Alpha 255	→	Notification text
duration	Number	→	Display duration in seconds

Note: This command only works under Windows.

Description

The DISPLAY NOTIFICATION command displays a message in the notification area of the Windows taskbar:

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.).

In title and text, pass the title and the text of the message to display (in the above example, the title is “4D Export”). You can enter up to 255 characters.

By default, the message window remains displayed until the user clicks on the close box. If you pass the optional duration parameter, the window will be closed automatically at the end of the duration set if the user did not click on the close box. Note that the notification icon will remain displayed until the end of duration, even if the user has closed the window.

See Also

ALERT.

GOTO XY (x; y)

Parameter	Type		Description
x	Number	→	x (horizontal) position of cursor
y	Number	→	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.

Tip: Using a fixed-width (monospaced) font, such as Terminal on Windows and Monaco on Macintosh, for the message, gives the best display results with GOTO XY and MESSAGE. See the description of the MESSAGE command for more information.

Examples

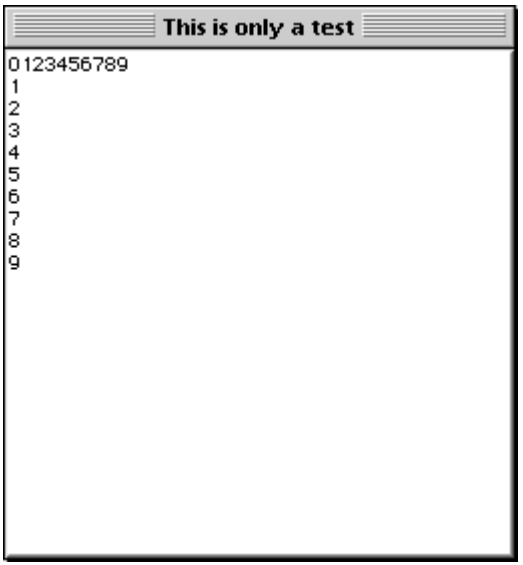
1. See example for the command MESSAGE.

2. See example for the command Milliseconds.

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:



See Also

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:

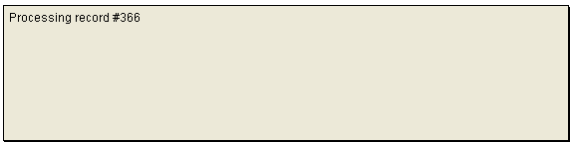
- Successive messages do not erase previous messages when displayed in the window. Instead, they are concatenated onto existing messages.
- If a message is wider than the window, 4D automatically performs text wrap.
- If a message has more lines than the window, 4D automatically scrolls the message window.
- 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.

Examples

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
```

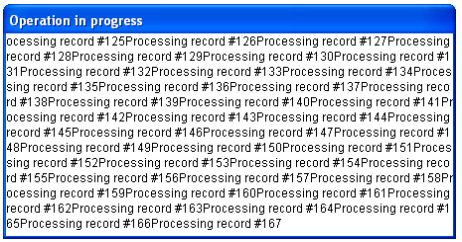
The following window appears and disappears at each MESSAGE call:



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
```

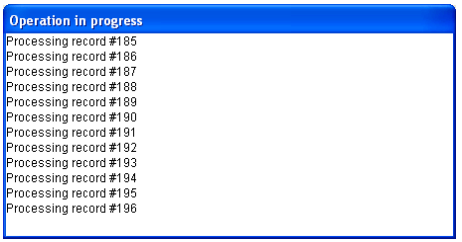
This provides the following result (shown here on Windows):



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
```

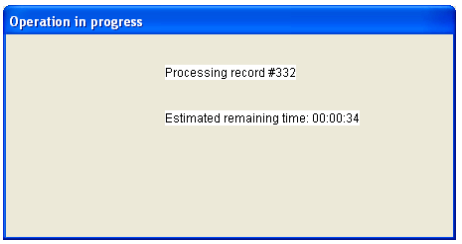
This provides the following result (shown here on Windows):



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):



See Also

CLOSE WINDOW, ERASE WINDOW, GOTO XY, Open window.

MESSAGES OFF

Parameter	Type	Description

This command does not require any parameters

Description

The MESSAGES ON and MESSAGES OFF 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	IMPORT SYLK	QUERY
DISTINCT VALUES	IMPORT TEXT	QUERY BY FORMULA
EXPORT DIF	RELATE MANY SELECTION	QUERY BY EXAMPLE
EXPORT SYLK	RELATE ONE SELECTION	QUERY SELECTION
EXPORT TEXT	REDUCE SELECTION	QUERY SELECTION BY FORMULA
GRAPH TABLE	QR REPORT	ORDER BY FORMULA
IMPORT DIF	SCAN INDEX	ORDER BY

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

Parameter	Type	Description

This command does not require any parameters

Description

See the description of the MESSAGES OFF command.

Request (message{; defaultResponse{; OKButtonTitle{; CancelButtonTitle}}}) → String

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. 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 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.

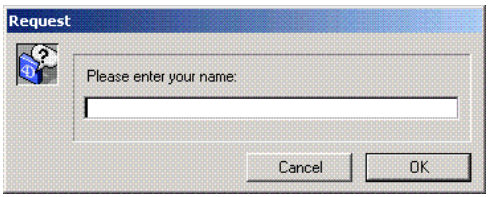
Tip: If you need to get several pieces of information from the user, design a form and present it with DIALOG, rather than presenting a succession of Request dialog boxes.

Examples

1. The line:

```
$vsPrompt:= Request ("Please enter your name:")
```

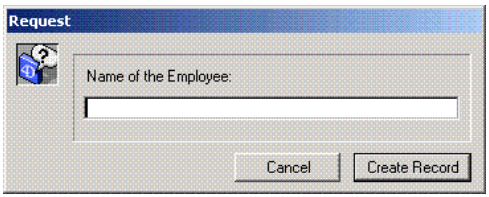
will display the request dialog box (on Windows) shown here:



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
```

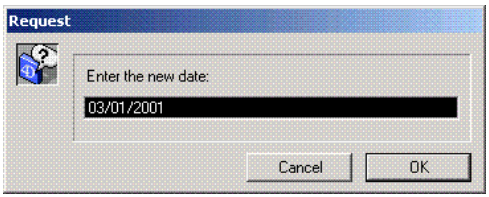
will display the request dialog box (on Windows) shown here:



3. The line:

```
$vdPrompt:= Date (Request ("Enter the new date:";String (Current date)))
```

will display the request dialog box (on Windows) shown here:



See Also

ALERT, CONFIRM.

29

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 process or interprocess in scope.

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).

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.

A named selection whose name is not prefixed with the symbols (<>) is process in scope and is available only within the process in which it was created.

With 4D Client 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.

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.

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.

See Also

Identifiers, SET QUERY DESTINATION.

CLEAR NAMED SELECTION

version 3

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. CLEAR NAMED SELECTION 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.

See Also

COPY NAMED SELECTION, CUT NAMED SELECTION, USE 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 Table 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")
```

See Also

CLEAR NAMED SELECTION, CUT NAMED SELECTION, Identifiers, USE NAMED SELECTION.

CREATE SELECTION FROM ARRAY (aTable; recordArray{; selectionName})

Parameter	Type		Description
aTable	Table	→	Table from which to create the selection
recordArray	Longint | Bool. 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.

See Also

CLEAR NAMED SELECTION, COPY NAMED SELECTION, CREATE SET FROM ARRAY, Identifiers, LONGINT ARRAY FROM SELECTION, USE 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")
```

See Also

CLEAR NAMED SELECTION, COPY NAMED SELECTION, Identifiers, 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.

See Also

COPY NAMED SELECTION, CUT NAMED SELECTION, USE NAMED SELECTION.

30

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 Properties 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.

Note: It is possible to 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.

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 second syntax is compatible with the previous version of 4D.

BEST OBJECT SIZ E ({*; }object; bestWidth; bestHeight{; maxWidth})

Parameter	Type		Description
*		→	If specified = object is an object name (String) If omitted = object is a variable
object	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 BEST OBJECT 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 MOVE OBJECT 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. Of course, 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. 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, BEST OBJECT 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.

For all other form object types (group areas, tabs, rectangles, straight lines, circles/ovals, external areas, etc.), the BEST OBJECT SIZE command returns the current object size (defined in the form editor and possibly using the MOVE OBJECT command).

Example

Refer to the example in the SET PRINT MARKER command.

See Also

MOVE OBJECT.

BUTTON TEXT ({*; }object; buttonText)

Parameter	Type		Description
*		→	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)
buttonText	String	→	New title for the button

Description

The BUTTON TEXT command changes the title of the buttons specified by object to the value you pass in buttonText.

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.

BUTTON TEXT affects only buttons that display text: buttons, check boxes, and radio buttons.

Usually, you will apply this command to one button at a time. The button area must be large enough to accommodate the text; if it is not, the text is truncated. Do not use carriage returns in buttonText.

Example

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
        BUTTON TEXT (bDelete;" Delete")
        DISABLE BUTTON (bDelete)
```

```
        : (Records in selection ([People]) = 1) ` One person found
            BUTTON TEXT (bDelete;"Delete Person")
            ENABLE BUTTON (bDelete)
        : (Records in selection([People]) > 1) ` Many people found
            BUTTON TEXT (bDelete;"Delete People")
            ENABLE BUTTON (bDelete)
    End case
```

See Also

DISABLE BUTTON, ENABLE BUTTON.

DISABLE BUTTON ({*; }object)

Parameter	Type		Description
*		→	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 DISABLE BUTTON command disables the form objects specified by object.

A disabled button or object does not react to mouse clicks and shortcuts, and is displayed dimmed or grayed out.

Note: Disabling a button or an object does not prevent you from changing its value programmatically.

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.

This command (despite what its name suggests) can be applied to the following types of object:

- Button, Default Button, 3D Button, Invisible Button, Highlight Button
- Radio Button, 3D Radio Button, Radio Picture
- 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 that is assigned an automatic action (4D changes the state of the control when needed), except for *Validate* and *Cancel* actions.

Examples

1. This example disables the button bValidate:

```
DISABLE BUTTON(bValidate)
```

2. This example disables all form objects that have names containing “btn”:

```
DISABLE BUTTON(*;"@btn@")
```

3. See example for the command BUTTON TEXT.

See Also

BUTTON TEXT, ENABLE BUTTON.

ENABLE BUTTON ({*; }object)

Parameter	Type		Description
*		→	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 ENABLE BUTTON command enables the form objects specified by object.

An enabled button or object reacts to mouse clicks and shortcuts.

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.

This command (despite what its name suggests) can be applied to the following types of object:

- Button, Default Button, 3D Button, Invisible Button, Highlight Button
- Radio Button, 3D Radio Button, Radio Picture
- 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 that is assigned an automatic action (4D changes the state of the control when needed), except for *Validate* and *Cancel* actions.

Examples

1. This example enables the button bValidate:

```
ENABLE BUTTON(bValidate)
```

2. This example enables all form objects that have names containing “btn”:

```
ENABLE BUTTON(*;"@btn@")
```

3. See example for the command BUTTON TEXT.

See Also

BUTTON TEXT, DISABLE BUTTON.

FONT ({*; }object; font)

Parameter	Type		Description
*		→	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 | Number	→	Font name or Font number

Description

FONT sets the form objects specified by object to be displayed using the font whose name or number you pass in font.

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.

Examples

1. The following example sets the font for a button named bOK:

```
FONT (bOK; "Arial")
```

2. The following example sets the font for all the form objects whose name contains "info":

```
FONT (*;"@info@"; "Times")
```

3. The following example assigns the special %password font, which can be used for entry and display of “password” type fields (characters are hidden).

```
FONT ([Users]Password"%password")
```

See Also

FONT SIZE, FONT STYLE.

FONT SIZE ({*; }object; size)

Parameter	Type		Description
*		→	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	Number	→	Font size in points

Description

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 section Object Properties.

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.

Examples

1. The following example sets the font size for a variable named vtInfo:

```
FONT SIZE (vtInfo; 14)
```

2. The following example sets the font size for all the form objects whose name starts with "hl":

```
FONT SIZE (*;"hl@"; 14)
```

See Also

FONT, FONT STYLE.

FONT STYLE ({*; }object; styles)

Parameter	Type		Description
*		→	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	Number	→	Font style

Description

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 section Object Properties.

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
Plain	Long Integer	0
Bold	Long Integer	1
Italic	Long Integer	2
Underline	Long Integer	4

Examples

1. This example sets the font style for a button named bAddNew. The font style is set to bold italic:

```
FONT STYLE (bAddNew; Bold + Italic)
```

2. This example sets the font style to Plain for all form objects with names starting with “vt”:

```
FONT STYLE (*;"vt@"; Plain)
```

See Also

FONT, FONT SIZE, SET LIST ITEM PROPERTIES.

Get alignment ({*; }object) → Number

Parameter	Type		Description
*		→	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	Number	←	Alignment code

Description

The Get alignment command returns a code indicating the type of 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 Object alignment theme:

Constant	Type	Value
Align default	Longint	1
Align left	Longint	2
Center	Longint	3
Align right	Longint	4

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

See Also

SET ALIGNMENT.

Get format ({*; }object) → String

Parameter	Type		Description
*	*	→	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 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 SET FORMAT command. 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 and 3D buttons. For more information on the display formats of these objects, refer to the documentation for the 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 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).

Examples

1. This example allows you to obtain the value of the format constant applied to the picture variable named “myphoto”:

```
C_STRING(2;$format)
SET FORMAT(*;"myphoto";Char(On background))
    `Apply background format (value = 3)
$format:=Get format(*;"myphoto")
ALERT("Format number:"+String(Character code($format)))
    `Display value "3"
```

2. This example allows you to obtain the format applied to the Boolean field [Members]Marital_status:

```
C_STRING(30;$format)
$format:=Get format([Members]Marital_status)
ALERT($format)  `Display format, for example "Married;Single"
```

See Also

SET FORMAT.

GET OBJECT RECT ({*; }object; left; top; right; bottom)

Parameter	Type		Description
*	*	→	If specified = object is the name of the object (string) If omitted = object is a variable
object	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 GET OBJECT RECT 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.

Example

Let's assume that you want to obtain the coordinates of a rectangle formed by all the objects that begin with "button":

```
GET OBJECT RECT(*;"button@";left;top;right;bottom)
```

See Also

MOVE OBJECT.

MOVE OBJECT ({*; }object; moveH; moveV{; resizeH{; resizeV{; *}}})

Parameter	Type		Description
*	*	→	If specified= object is an object name (string) If omitted = object is a variable
object	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
*	*	→	If specified = absolute coordinates If omitted = relative coordinates

Description

The MOVE OBJECT 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.

Examples

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:

```
MOVE OBJECT (*;"button_1";10;-20;30;40)
```

2. The following statement moves “button_1” to the following coordinates (10;20) (30;40):

```
MOVE OBJECT (*;"button_1";10;20;30;40;*)
```

See Also

GET OBJECT RECT.

SET ALIGNMENT ({*; }object; alignment)

Parameter	Type		Description
*		→	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	Number	→	Alignment code

Description

The SET ALIGNMENT command allows you to set the type of 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 Object alignment theme in the alignment parameter:

Constant	Type	Value
Align default	Longint	1
Align left	Longint	2
Center	Longint	3
Align right	Longint	4

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

See Also

Get alignment.

SET CHOICE LIST ({*; }object; list)

Parameter	Type		Description
*		→	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)
list	String	→	Name of the list to use as Choice list (as defined in Design environment)

Description

The SET CHOICE LIST command sets or replaces the choice list associated with the object or group of objects specified by object to the choice list (defined in the Design environment List Editor) whose name you pass in list.

This command can be applied in an input or dialog form, to fields and enterable variables whose value can be entered as text. The choice list is displayed during data entry when the user selects the text area.

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.

Note: This command cannot be used with fields located in a subform's list form.

Example

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)
    SET CHOICE LIST([Shipments]Shipper; "Fast Shippers")
Else
    SET CHOICE LIST([Shipments]Shipper; "Normal Shippers")
End if
```

SET COLOR ({*; }object; color{; altColor})

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
object	Field or variable	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
color	Number	→	New colors for the object
altColor	Number	→	Alternating colors for a list box

Description

The 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 section Object Properties.

altColor is used to specify an alternative color for the even-numbered rows of a list box or a list box column. 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.

The color (as well as altColor) 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.

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
White	Long Integer	0
Yellow	Long Integer	1
Orange	Long Integer	2
Red	Long Integer	3
Purple	Long Integer	4
Dark Blue	Long Integer	5
Blue	Long Integer	6
Light Blue	Long Integer	7
Green	Long Integer	8
Dark Green	Long Integer	9
Dark Brown	Long Integer	10
Dark Grey	Long Integer	11
Light Grey	Long Integer	12
Brown	Long Integer	13
Grey	Long Integer	14
Black	Long Integer	15

Note: While SET COLOR works with indexed colors within the default 4D color palette, the command SET RGB COLORS allows you to work with any RGB color. To reestablish automatic colors for an object, use the SET RGB COLORS command with the Default foreground color and Default background color constants.

Example

The following example sets the color of the text area shown below in the form editor:

After executing the following statement:

```
SET COLOR (*;"Mytext"; - (Yellow + (256 * Red)))
```

... the area appears as follows:

See Also

SET RGB COLORS.

SET ENTERABLE ({*; }entryArea; enterable)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
entryArea	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 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 section Object Properties.

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 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: SET ENTERABLE(*;"Subform";True).
• For list forms, you must pass the name of the form table in the entryArea parameter, for example: SET ENTERABLE([MyTable];True).

Making an object non-enterable does not prevent you from changing its value programmatically.

Examples

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"
    SET ENTERABLE([Shipments]Shipper;False)
Else
    SET ENTERABLE([Shipments]Shipper;True)
End if
```

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)
SET ENTERABLE([Table1];bEnterable)
```

See Also

DISABLE BUTTON, ENABLE BUTTON, SET VISIBLE.

SET FILTER ({*; }object; entryFilter)

Parameter	Type		Description
*		→	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

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 section Object Properties.

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 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 (|).

Examples

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
    SET FILTER ([Companies]ZIP Code; "&9#####")
Else ` Set the filter to accept alpha and numeric and uppercase the alpha
    SET FILTER ([Companies]ZIP Code; "~@")
End if
```

2. The following example allows only the letters “a,” “b,” “c,” or “g” to be entered in two places in the field Field:

```
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"##.

See Also

SET FORMAT.

SET FORMAT ({*; }object; displayFormat)

Parameter	Type		Description
*		→	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

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 section Object Properties.

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
System date short	Long Integer	1
System date abbreviated	Long Integer	2
System date long	Long Integer	3
Internal date short special	Long Integer	4
Internal date long	Long Integer	5
Internal date abbreviated	Long Integer	6
Internal date short	Long Integer	7
ISO Date Time	Long Integer	8
Blank if null	Long Integer	100

Note: The Blank if null 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
HH MM SS	Long Integer	1
HH MM	Long Integer	2
Hour Min Sec	Long Integer	3
Hour Min	Long Integer	4
HH MM AM PM	Long Integer	5
MM SS	Long Integer	6
Min Sec	Long Integer	7
ISO Date Time	Long Integer	8
System time short	Long Integer	9
System time long abbreviated	Long Integer	10
System time long	Long Integer	11
Blank if null	Long Integer	100

Note: The Blank if null 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
Truncated Centered	Long Integer	1
Scaled to Fit	Long Integer	2
On Background	Long Integer	3
Truncated non Centered	Long Integer	4
Scaled to fit proportional	Long Integer	5
Scaled to fit prop centered	Long Integer	6
Replicated	Long Integer	7

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 *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, a picture variable or a PICT resource:
- 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.
- If the picture comes from a PICT resource, enter its number, preceded by a colon (e.g.: ":62500").

• 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, a picture variable or a PICT resource:
- 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.
- If the picture comes from a PICT resource, enter its number, preceded by a colon (e.g.: ":62500")
• 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:
- mode = 0: the picture pop-up menu is not transparent,
- mode = 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}

• 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. 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.

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

• 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, a picture variable or a PICT resource:
- 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 PICT resource, enter its number, preceded by a colon (ex.: ":62500").
- If the picture comes from a fiole 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). Ten 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
• 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

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 and vertMargin options, you can write:

```
SET FORMAT(myVar;"NiceButton;?256;:562;1;;1;4;5;;5;0")
```

Examples

1. The following line of code formats the [Employee]Date Hired field to Month Date Year.

```
SET FORMAT ([Employee]Date Hired; Char(Month Date Year))
```

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)
    SET FORMAT ([Company]ZIP Code; "#####–####")
Else
    SET FORMAT ([Company]ZIP Code; "#####")
End if
```

3. The following example sets the format of a Boolean field to display Married and Unmarried, instead of the default Yes and No:

```
SET FORMAT ([Employee]Marital Status;"Married;Unmarried")
```

4. The following example sets the format of a Boolean field to display a checkbox labelled “Classified”:

```
SET FORMAT ([Folder]Classification; "Classified")
```

5. 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:

```
SET FORMAT (*;"PictureButton"; "4;1;?15000;176")
```

6. Switching a thermometer to "Barber shop" mode:

```
SET FORMAT ($Mythermo;";;;;128")
$Mythermo:=1 `Start animation
```

See Also

Get format, GET SYSTEM FORMAT, SET FILTER.

SET RGB COLORS ({*; }object; foregroundColor; backgroundColor{; altBackgrndColor})

Parameter	Type		Description
*		→	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	Number	→	RGB color value for Foreground color
backgroundColor	Number	→	RGB color value for Background color
altBackgrndColor	Number	→	RGB color value for Alternating background color

Description

The SET RGB COLORS command changes the foreground and background colors of the objects specified by object and the optional * parameters. 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 section Object Properties.

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)

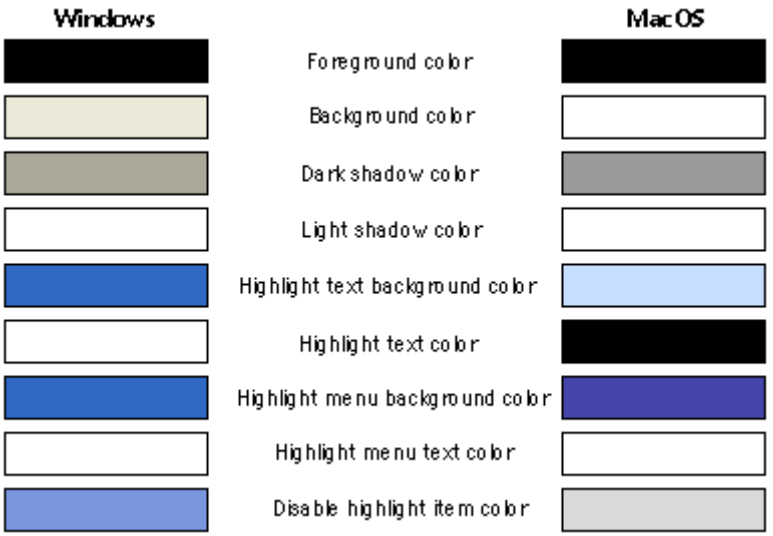
The following table shows some examples of RGB color values:

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
Foreground color	Long Integer	-1
Background color	Long Integer	-2
Dark shadow color	Long Integer	-3
Light shadow color	Long Integer	-4
Highlight text background color	Long Integer	-7
Highlight text color	Long Integer	-8
Highlight menu background color	Long Integer	-9
Highlight menu text color	Long Integer	-10
Disable highlight item color	Long Integer	-11

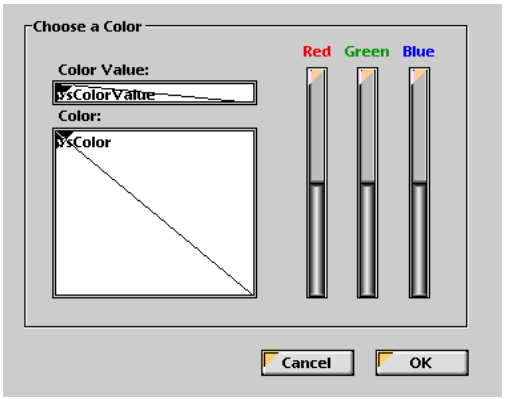
These colors (on a standard system) are shown here:



WARNING: These automatic colors are system dependent. If you change 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.

Examples

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:

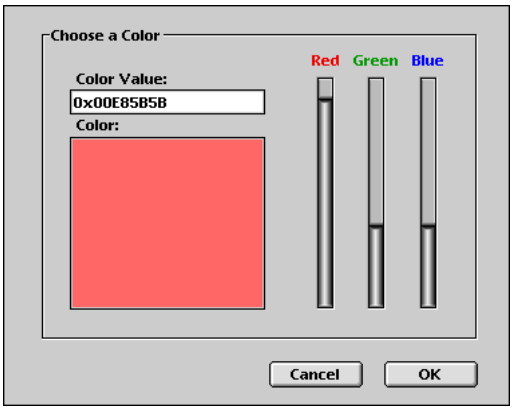
```
    ` vsColorValue non-enterable Object Method
Case of
    : (Form event=On Load)
        vsColorValue:="0x00000000"
End case
    ` vsColor non-enterable variable Object Method
Case of
    : (Form event=On Load)
        vsColor:=""
        SET RGB COLORS(vsColor;0x00FFFFFF;0x0000)
End case

    ` 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
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:



See Also

Bitwise Operators, Select RGB Color, SET COLOR.

SET SCROLLBAR VISIBLE ({*; }object; horizontal; vertical)

Parameter	Type		Description
*		→	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 SET SCROLLBAR VISIBLE 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.

This command is used with the following form objects:
- list boxes,
- scrollable areas,
- hierachical lists,
- subforms.

Pass the Boolean values in horizontal and vertical indicating whether the corresponding scrollbars should be displayed (True) or hidden (False). The scrollbars are displayed by default.

Note: Objects of the scrollable area type do not have horizontal scrollbars. Since the horizontal parameter is mandatory, you must still pass it in this case; however, it will be ignored.

See Also

Get listbox information, SET VISIBLE, SHOW LISTBOX GRID.

SET VISIBLE ({*; }object; visible)

Parameter	Type		Description
*		→	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

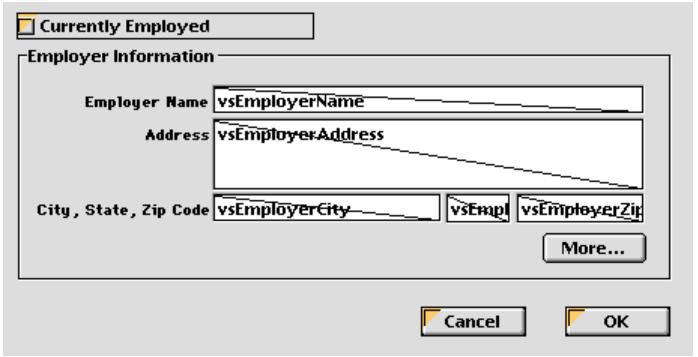
The SET VISIBLE command shows or hides the objects specified by object.

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

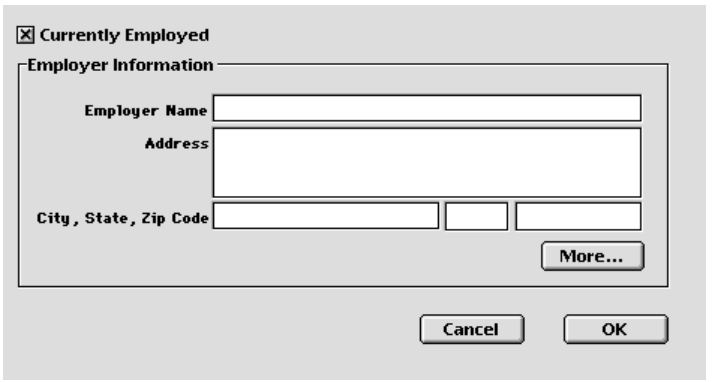
Here is a typical form in the Design environment:



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"
        SET VISIBLE(*;"@emp@";cbCurrentlyEmployed # 0)
            ` But always keep the check box itself visible
        SET VISIBLE(cbCurrentlyEmployed;True)
End case
```

Therefore, when executed, the form looks like:



or:

See Also

DISABLE BUTTON, ENABLE BUTTON, SET ENTERABLE.

31

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 can be applied to fields or subfields. In the case of a field, they are applied to a selection of records. In the case of a subfield, they are applied to a selection of the subrecords of the current record. Note that the
Sum squares, Std deviation, and Variance functions can be used on a field only during printing.

These functions work on numeric data only. Each of these functions returns a numeric value.

Using a field

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 avoid this, index the field.

When these functions are used in a report, they behave differently than at other times. This is 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.

See Also

Average, Max, Min, Std deviation, Sum, Sum squares, Variance.

Average (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the average
Function result	Number	←	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.

Example

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)
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.

See Also

ACCUMULATE, BREAK LEVEL, Max, Min, ORDER BY, PRINT SELECTION, Subtotal, Sum.

Max (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the maximum value
Function result	Number	←	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.

If the series selection is empty, Max returns -1E50.

Example

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)
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.

See Also

Min.

Min (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the minimum value
Function result	Number	←	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, Max returns 1E50.

Examples

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)
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.

2. The following example finds the lowest sale amount of an employee and displays the result in an alert box. The sales amounts are stored in the subfield [Employees]SalesDollars:

```
ALERT ("Minimum sale = " + String(Min([Employees]SalesDollars)))
```

See Also

Max.

Std deviation (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the standard deviation
Function result	Number	←	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 only use a field with this function when printing a report.

Example

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.

See Also

Average, Sum, Sum squares, Variance.

Sum (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the sum
Function result	Number	←	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.

Example

The following example is an object method for a variable that vTotal 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.

See Also

ACCUMULATE, Average, BREAK LEVEL, Max, Min, ORDER BY, PRINT SELECTION, Subtotal.

Sum squares (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the sum of squares
Function result	Number	←	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 only use a field with this function when printing a report.

Example

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 printing commands.

See Also

Average, Std deviation, Sum, Variance.

Variance (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the variance
Function result	Number	←	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 only use a field with this function when printing a report.

Example

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 printing commands.

See Also

Average, Std deviation, Sum, Sum squares.

32

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 section String Operators.

Numeric Operators
See the section Numeric Operators.

Date Operators
See the section Date Operators.

Time Operators
See the section Time Operators.

Comparison Operators
See the section Comparison Operators.

Logical Operators
See the section Logical Operators.

Picture Operators
See the section Picture Operators.

Bitwise Operators
See the section Bitwise Operators.

See Also

Constants, Data Types, Identifiers, QUERY, QUERY BY FORMULA, QUERY SELECTION BY FORMULA.

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 32. 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 expression returns the value of the first operand unchanged for Bit Set and Bit Clear, and returns False for Bit Test.

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. Here is the logical AND table: 1 & 1 → 1 0 & 1 → 0 1 & 0 → 0 0 & 0 → 0 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. Here is the logical OR table: 1 | 1 → 1 0 | 1 → 1 1 | 0 → 1 0 | 0 → 0 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. Here is the logical XOR table: 1 ^| 1 → 0 0 ^| 1 → 1 1 ^| 0 → 1 0 ^| 0 → 0 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. 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. 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.

Examples

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
System heap resource mask	Long Integer	64
System heap resource bit	Long Integer	6
Purgeable resource mask	Long Integer	32
Purgeable resource bit	Long Integer	5

Locked resource mask	Long Integer	16
Locked resource bit	Long Integer	4
Protected resource mask	Long Integer	8
Protected resource bit	Long Integer	3
Preloaded resource mask	Long Integer	4
Preloaded resource bit	Long Integer	2
Changed resource mask	Long Integer	2
Changed resource bit	Long Integer	1

These constants enable you to test the value returned by Get resource properties or to create the value passed to 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.

See Also

Comparison Operators, Date Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, String Operators, Time Operators.

The tables in this section show the comparison operators as they apply to string, numeric, date, time, and pointer expressions. An expression that uses a comparison operator returns a Boolean value, either TRUE or FALSE.

String Comparisons

Operation	Syntax	Returns	Expression	Value
Equality	String = String	Boolean	"abc" = "abc" "abc" = "abd"	True False
Inequality	String # String	Boolean	"abc" # "abd" "abc" # "abc"	True False
Greater than	String > String	Boolean	"abd" > "abc" "abc" > "abc"	True False
Less than	String < String	Boolean	"abc" < "abd" "abc" < "abc"	True False
Greater than or equal to	String >= String	Boolean	"abd" >= "abc" "abc" >= "abd"	True False
Less than or equal to	String <= String	Boolean	"abc" <= "abd" "abd" <= "abc"	True False
Contains keyword	String % String	Boolean	"Alpha Bravo" % "Bravo" "Alpha Bravo" % "ravo"	True False

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 10 = 11	True False
Inequality	Number # Number	Boolean	10 #11 10 # 10	True False
Greater than	Number > Number	Boolean	11 > 10 10 > 11	True False
Less than	Number < Number	Boolean	10 < 11 11 < 10	True False
Greater than or equal to	Number >= Number	Boolean	11 >= 10 10 >= 11	True False
Less than or equal to	Number <= Number	Boolean	10 <= 11 11 <= 10	True False

Date Comparisons

Operation	Syntax	Returns	Expression	Value
Equality	Date = Date	Boolean	!1/1/97! =!1/1/97! !1/20/97! =!1/1/97!	True False
Inequality	Date # Date	Boolean	!1/20/97! # !1/1/97! !1/1/97! # !1/1/97!	True False
Greater than	Date > Date	Boolean	!1/20/97! > !1/1/97! !1/1/97! > !1/1/97!	True False
Less than	Date < Date	Boolean	!1/1/97! < !1/20/97! !1/1/97! < !1/1/97!	True False
Greater than or equal to	Date >= Date	Boolean	!1/20/97! >=!1/1/97! !1/1/97!>=!1/20/97!	True False
Less than or equal to	Date <= Date	Boolean	!1/1/97!<=!1/20/97! !1/20/97!<=!1/1/97!	True 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. Numbers can be searched for because they are evaluated as strings; however, decimal separators (. ,)and 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
```

• 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"
```

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 character for Query and Order By" option which can be accessed using the Preferences 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.

See Also

Bitwise Operators, Date Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, Time Operators.

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	!1/20/97! – !1/1/97!	19
Day addition	Date + Number	Date	!1/20/97! + 9	!1/29/97!
Day subtraction	Date – Number	Date	!1/20/97! – 9	!1/11/97!

See Also

Bitwise Operators, Comparison Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, String Operators, Time 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)

See Also

Bitwise Operators, Comparison Operators, Date Operators, Numeric Operators, Operators, Picture Operators, String Operators, Time 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

Modulo Operator

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.

See Also

Bitwise Operators, Comparison Operators, Date Operators, Logical Operators, Operators, Picture Operators, String Operators, Time 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	Perform an XOR on Pict1 and Pict2
Inclusive superimposition	Pict1 | Pict2	Perform a OR on Pict1 and Pict2
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 ratio	Picture *+ Number	Resize Picture horizontally by Number
Vertical scaling	Picture */ Number	Resize Picture vertically by Number ratio

The two operators & and | always return a bitmapped picture, no matter what the nature of the two source pictures. The reason is that 4D first draws the pictures into memory bitmaps, then calculates the resulting picture by performing graphical exclusive or inclusive OR on the pixels of the bitmaps.

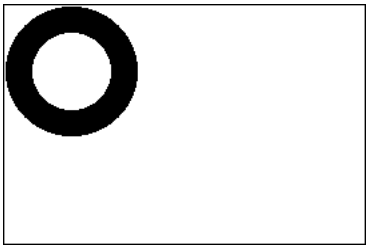
Note: The COMBINE PICTURES command can be used to superimpose pictures while keeping the characteristics of each source picture in the resulting picture.

The other 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.

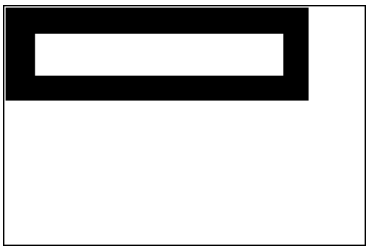
Examples

In the following examples, all of the pictures are shown using the display format On Background.

Here is the picture circle:



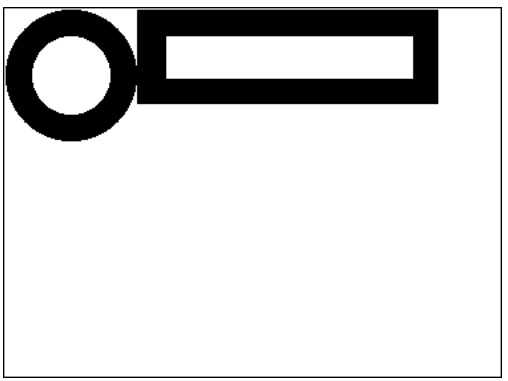
Here is the picture rectangle:



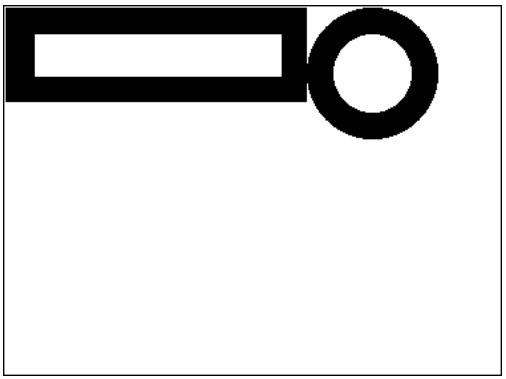
In the following examples, each expression is followed by its graphical representation.

• Horizontal concatenation

```
circle + rectangle ` Place the rectangle to the right of the circle
```

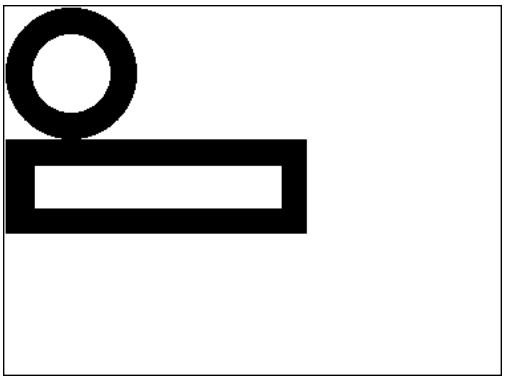


rectangle + circle ` Place the circle to the right of the rectangle

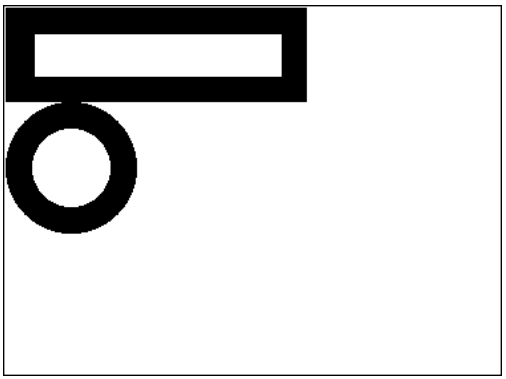


- Vertical concatenation

circle / rectangle ` Place the rectangle under the circle

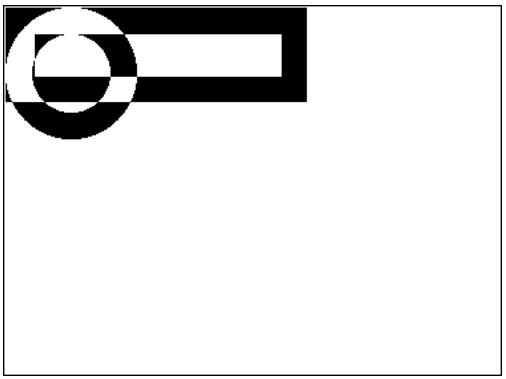


rectangle / circle ` Place the circle under the rectangle



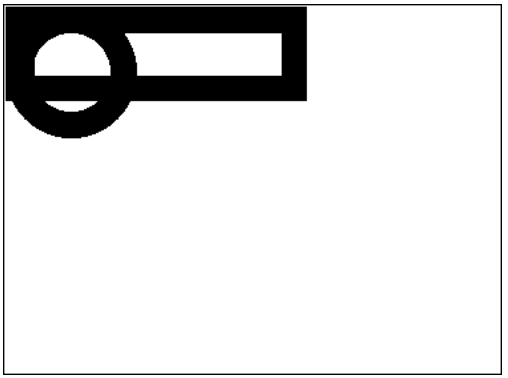
• Exclusive superimposition (XOR)

circle & rectangle ` Exclusive OR of the two pictures



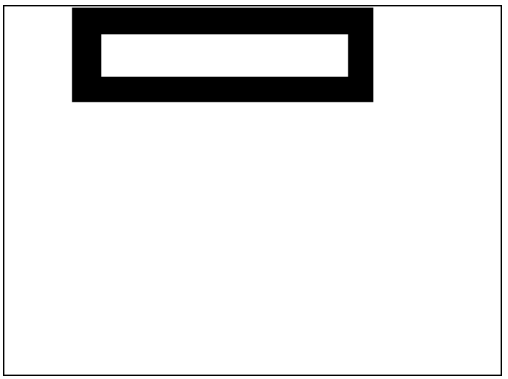
• Inclusive superimposition (OR)

```
circle | rectangle ` Inclusive OR of the two pictures
```

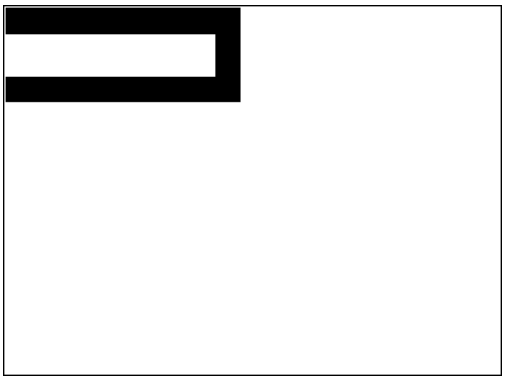


• Horizontal move

```
rectangle + 50 ` Move the rectangle 50 pixels to the right
```

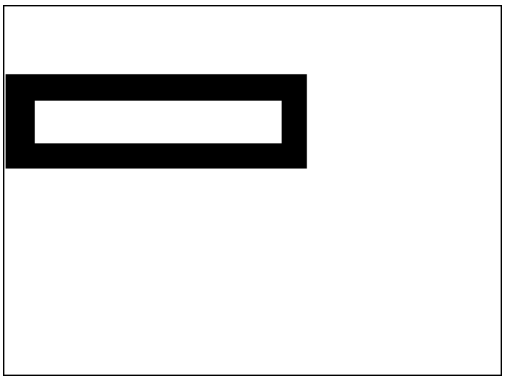


rectangle - 50 ` Move the rectangle 50 pixels to the left

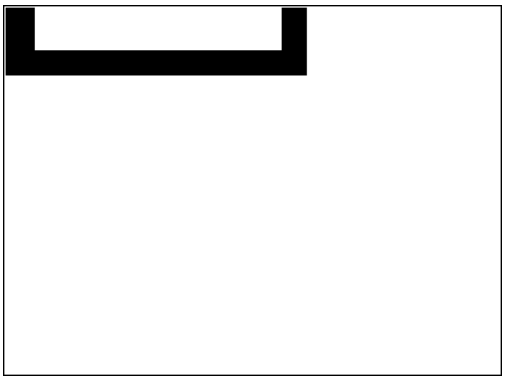


- Vertical move

rectangle /50 ` Move the rectangle down by 50 pixels



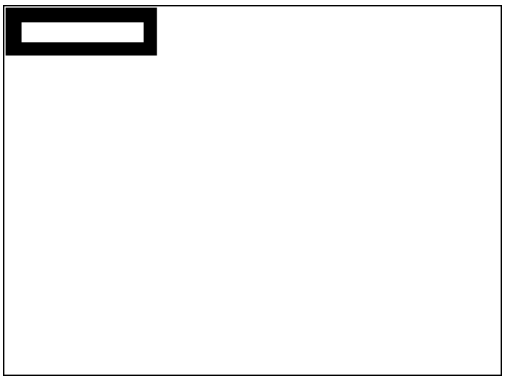
rectangle /-20 ` Move the rectangle up by 20 pixels



- Resize

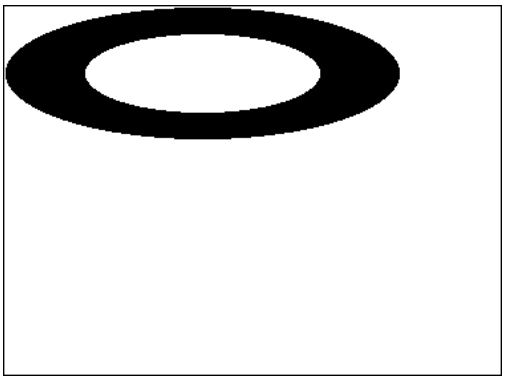
rectangle * 1.5 ` The rectangle becomes 50% bigger

rectangle * 0.5 ` The rectangle becomes 50% smaller

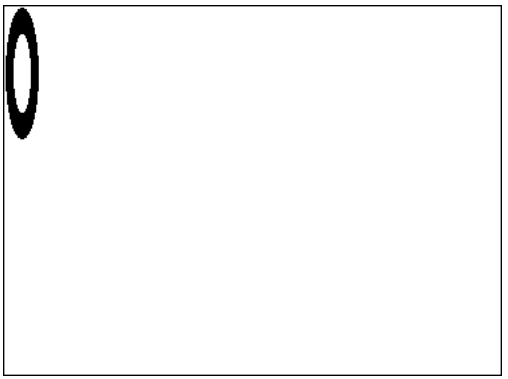


• Horizontal scaling

circle *+3 ` The circle becomes 3 times wider

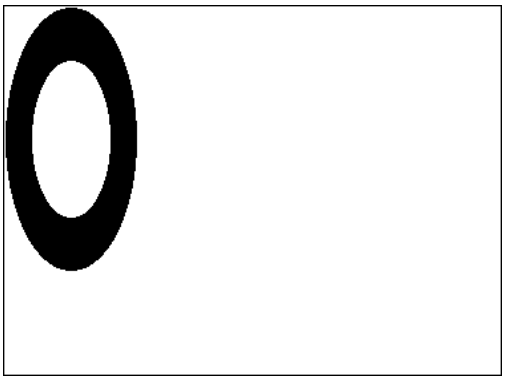


circle *+ 0.25 ` The circle's width becomes a quarter of what it was

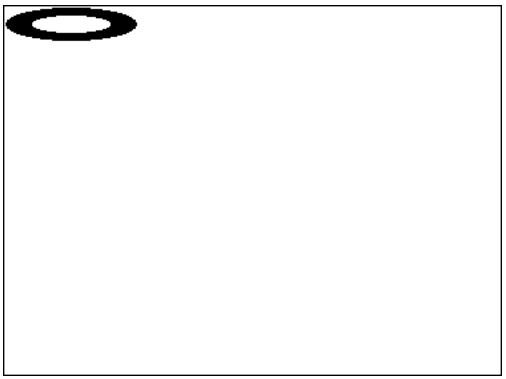


- Vertical scaling

circle */ 2 ` The circle becomes twice as tall



circle */ 0.25 ` The circle's height becomes a quarter of what it was



See Also

Bitwise Operators, COMBINE PICTURES, Comparison Operators, Date Operators, Logical Operators, Numeric Operators, Operators, String Operators, Time Operators, TRANSFORM PICTURE.

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	"abab"

See Also

Bitwise Operators, Comparison Operators, Date Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, 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 % Number	Number	?02:03:04? % 2	0

Tips

(1) To obtain a time expression from an expression that combines a time expression with a number, use the commands Time and Time string.

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(Time string(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?
```

However, while developing your application, you may encounter situations where a delay, expressed in seconds and added to a time value, is only available to you as a numeric value. In this case, use the next tip.

(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
```

See Also

Bitwise Operators, Comparison Operators, Date Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, String Operators.

33

Pasteboard

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.text.native"	Text in native character set
"com.4d.text.utf16"	Text in Unicode character set
"com.4d.text.rtf"	Enriched text
"com.4d.picture.pict"	PICT picture format
"com.4d.picture.pgn"	PGN picture format
"com.4d.picture.gif"	GIF picture format
"com.4d.picture.jfif"	JPEG picture format
"com.4d.picture.emf"	EMF picture format
"com.4d.picture.bitmap"	BITMAP picture format
"com.4d.picture.tiff"	TIFF picture format
"com.4d.picture.pdf"	PDF document
"com.4d.file.url"	File pathname

APPEND DATA TO PASTEBOARD

APPEND DATA TO PASTEBOARD (dataType; data)

Parameter	Type		Description
dataType	String	→	4-character data type string
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.

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

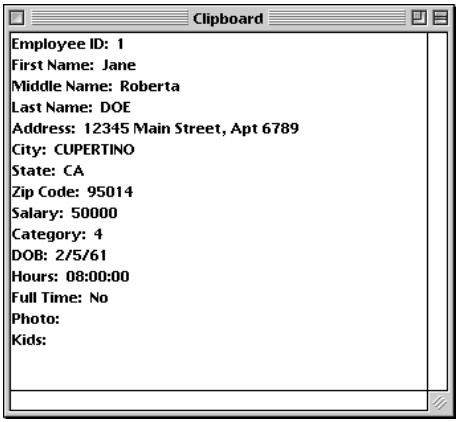
Using Pasteboard commands and BLOBs, you can build sophisticated Cut/Copy/Paste schemes that deal with structured data rather than a unique piece of data. In the following example, the two project methods SET RECORD TO PASTEBOARD and GET RECORD FROM PASTEBOARD enable you to treat a whole record as one 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 PASTEBOARDmethod
      ` 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
```

See Also

CLEAR PASTEBOARD, SET PICTURE TO PASTEBOARD, SET TEXT TO PASTEBOARD.

System Variables

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.

Error Handling

If there is not enough memory to append the BLOB data to the pasteboard, an error -108 is generated.

CLEAR PASTEBOARD

Parameter	Type	Description

This command 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.

Examples

1. The following code clears and then appends data to the pasteboard:

```
CLEAR PASTEBOARD  ` Make sure the pasteboard is emptied
   ` Add some gif pictures
APPEND DATA TO PASTEBOARD("com.4d.picture.gif";$vxSomeData)
APPEND DATA TO PASTEBOARD("com.4d.text.rtf";$vxSylkData)  ` Add some RTF text
```

2. See example for the APPEND DATA TO PASTEBOARD command.

See Also

APPEND DATA TO PASTEBOARD.

Get file from pasteboard

version 11

Get file from pasteboard (xIndex) → String

Parameter	Type		Description
xIndex	Number	→	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 comand 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="")
```

See Also

SET FILE TO PASTEBOARD.

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 into the BLOB field or into the variable data the data present in the pasteboard and whose type you pass in dataType.

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.

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?
        ` Accumulate the array elements in a BLOB
    VARIABLE TO BLOB (asOptions;$vxClipData)
    CLEAR PASTEBOARD ` Empty the pasteboard
        ` Note the data type arbitrarily chosen
    APPEND DATA TO PASTEBOARD ("artx";asOptions)
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
```

See Also

APPEND DATA TO PASTEBOARD, GET PICTURE FROM PASTEBOARD, Get text from pasteboard.

System Variables

If the data is correctly extracted, OK is set to 1; otherwise OK is set to 0 and an error is generated.

Error Handling

- If there is not enough memory to extract the data, an error -108 is generated.
- If there is no data of the requested type in the pasteboard, an error -102 is generated.

GET PASTEBOARD DATA TYPE

version 11

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 can be used to get a 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.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:
- UnderMac 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.

See Also

Managing Pasteboards.

GET PICTURE FROM PASTEBOARD

version 11 (Modified)

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 into the picture field or variable.

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.). When the destination area accepts native formats, the picture keeps its format; otherwise it is converted to the PICT format.

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.picture.jpeg")>0) | (Pasteboard data size
                                                    ("com.4d.picture.gif")>0))
   GET PICTURE FROM PASTEBOARD([Employees]Photo)
Else
   ALERT ("The pasteboard does not contain any pictures.")
End if
```

See Also

GET PASTEBOARD DATA, Get text from pasteboard, Pasteboard data size.

System Variables

If the picture is correctly extracted, OK is set to 1; otherwise OK is set to 0 and an error is generated.

Error Handling

- If there is not enough memory to extract the picture, an error -108 is generated.
- If there is no picture in the pasteboard, an error -102 is generated.

Get text from pasteboard

version 11 (Modified)

Get text from pasteboard → String

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

GET PASTEBOARD DATA, GET PICTURE FROM PASTEBOARD, Pasteboard data size.

System Variables

If the text is correctly extracted, OK is set to 1; otherwise OK is set to 0 and an error is generated.

Error Handling

- If there is not enough memory to extract the text, an error -108 is generated.
- If there is no text in the pasteboard, an error -102 is generated.

Pasteboard data size

version 11 (Modified)

Pasteboard data size (dataType) → Number

Parameter	Type		Description
dataType	String	→	Data type
Function result	Number	←	Size (in bytes) of data located in the pasteboard or error code

Description

The Pasteboard data size command allows you to check 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.

Examples

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) > 0)  ` Is there a picture in the pasteboard?
      ` If so, extract the picture from the pasteboard
   GET PICTURE FROM PASTEBOARD($vPicVariable)
Else
   ALERT("There is no picture in the pasteboard.")
End if
```

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.

3. See the example for the APPEND DATA TO PASTEBOARD command.

See Also

GET PASTEBOARD DATA, GET PICTURE FROM PASTEBOARD, Get text from pasteboard.

SET FILE TO PASTEBOARD

version 11

SET FILE TO PASTEBOARD (filePath)

Parameter	Type		Description
filePath	String	→	Complete pathname of file

Description

The SET FILE TO PASTEBOARD command adds the complete pathname passed in the filePath 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.

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.

See Also

Get file from pasteboard.

SET PICTURE TO PASTEBOARD

version 11 (Modified)

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.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
```

See Also

APPEND DATA TO PASTEBOARD, GET PICTURE FROM PASTEBOARD.

System Variables or 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 (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.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.

See Also

APPEND DATA TO PASTEBOARD, Get text from pasteboard.

System Variables or 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.

34

Pictures

Native Formats Supported

4D integrates native management of the most current picture formats, such as JPEG or GIF. 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.

The native formats are available in all cases and will always be returned by the PICTURE CODEC LIST command, regardless of the operating system and machine configuration. These formats are the following:

- Jpeg
- Png
- Bmp
- Gif
- Tif
- Emf (Windows only)
- Pict
- Pdf (Mac OS only)

Note: If 4D cannot interpret the picture format, the program calls on Quicktime routines (see below).

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 three different forms:

- As an extension (for example ".gif")
- As a Mime type (for example "image/jpeg")
- As a 4-character QuickTime code (for example "PNTG")

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.

Coordinates for Clicks on a Picture

4D lets you retrieve the local coordinates of a click on a picture field or variable, even if a scroll or zoom has been applied to the picture.
The click 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). You must get the value of these variables as part of the On Clicked or On Double Clicked form event. In order for this mechanism to work properly, the display format must be set to "Truncated non-centered" (see the SET FORMAT command).
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.

Using Apple QuickTime with 4D

4D can use Apple QuickTime routines to manage picture storage and display in databases. Under Mac OS, QuickTime is integrated into the operating system, no extension is required. Under Windows, 4D requires **QuickTime version 4 (or higher)** to be installed in order for you to be able to use picture compression/decompression on this platform.

Compatibility Note: The LOAD COMPRESS PICTURE FROM FILE, COMPRESS PICTURE FILE and COMPRESS PICTURE commands call upon obsolete mechanisms. They can be favorably replaced by the WRITE PICTURE FILE, PICTURE TO BLOB or CONVERT PICTURE commands. Moreover, commands that call on disk files (LOAD COMPRESS PICTURE FROM FILE and COMPRESS PICTURE FILE) will not work under Windows, no matter what version of QuickTime is installed.

• Picture Conversion and Compression Errors

When you try to use a picture conversion or compression command and QuickTime is not installed in your system, 4D returns the error code -9955. Other errors generated by QuickTime can also be returned. You can catch these errors using an error-handling method installed with ON ERR CALL.

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

version 6.7

BLOB TO PICTURE (pictureBlob; picture)

Parameter	Type		Description
pictureBlob	BLOB	→	BLOB containing a picture
picture	Picture	←	Picture from BLOB

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 or any QuickTime supported format. You can obtain the list of available formats using the PICTURE CODEC LIST command.

Pass in the picture parameter the 4D picture field or variable which should display the picture.

Note: The internal picture format is stored within the 4D variable or field. In the case of custom formats using QuickTime, it is necessary to get QuickTime to read the picture within 4D.

Once the command has been executed, the picture parameter contains the picture to display.

If the conversion has been done successfully, the system variable OK is set to 1. If the conversion has failed (QuickTime is not installed, or the BLOB does not contain a readable picture), OK is set to 0 and the picture variable or field is empty.

See Also

PICTURE CODEC LIST, PICTURE TO BLOB, PICTURE TYPE LIST, READ PICTURE FILE.

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 can be used to combine 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.

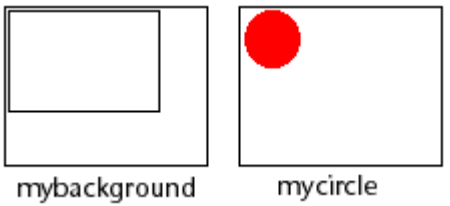
Note: This command extends the functionalities offered by the conventional picture combination operators (+/, etc., see the Picture Operators section). These operators remain entirely usable in 4D v11.

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.

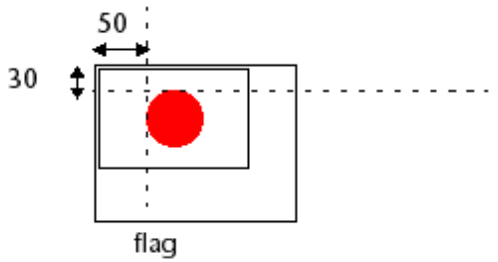
Example

Given the following pictures:



COMBINE PICTURES(flag;mybackground;Superimposition;mycircle;50;30)

Result:



See Also

Picture Operators, TRANSFORM PICTURE.

COMPRESS PICTURE

COMPRESS PICTURE (picture; method; quality)

Parameter	Type		Description
picture	Picture	→	Picture to be compressed
		←	Compressed picture
method	String	→	4-character string compression method
quality	Number	→	Compression quality (1..1000)

Compatibility note: This command calls for obsolete mechanisms and is only kept for compatibility reasons. It has been favorably replaced by the CONVERT PICTURE command.

Description

The COMPRESS PICTURE command compresses the picture contained in the field or variable picture.

The parameter method is a 4-character string indicating the compressor type. You should pass one of the constants of the Picture compression theme in this parameter.

The parameter quality is an integer between 1 and 1000 indicating the quality of the compressed picture. In general, reducing the quality will allow for greater compression of the picture.

Warning: The compression ratio possible for a given quality depends on the size and nature of the picture you are compressing. Compressing small pictures may not produce any decrease in size.

See Also

COMPRESS PICTURE FILE, CONVERT PICTURE, LOAD COMPRESS PICTURE FROM FILE, Pictures.

COMPRESS PICTURE FILE

version 3

COMPRESS PICTURE FILE (document; method; quality)

Parameter	Type		Description
document	DocRef	→	Document reference number
method	String	→	4-character string compression method
quality	Number	→	Compression quality (1..1000)

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 or PICTURE TO BLOB commands.

Description

This command compresses a picture document on disk. Use this command to compress a picture that you know cannot be loaded with the available memory. Once compressed, it can be loaded into memory using LOAD COMPRESS PICTURE FROM FILE.

Note: This command does not work on Windows.

The parameter method is a 4-character string indicating the compressor type. You should pass one of the constants of the Picture compression theme in method.

The parameter quality is an integer between 1 and 1000 indicating the quality of the compressed picture. In general, reducing the quality will allow for greater compression of the picture.

Warning: The compression ratio possible for a given quality depends on the size and nature of the picture you are compressing. Compressing small pictures may not produce any decrease in size.

Example

The following example presents the Open File dialog box that allows you to select a PICT file. Only PICT files will be displayed. The picture is compressed, loaded into memory, and stored in a picture variable. The file is then closed.

```
vRef:=Open document ("";"PICT")
If (OK=1)
    COMPRESS PICTURE FILE(vRef;QT Photo compressor;500)
    LOAD COMPRESS PICTURE FROM FILE(vRef;"";500;vPict)
    CLOSE DOCUMENT(vRef)
End if
```

See Also

COMPRESS PICTURE, LOAD COMPRESS PICTURE FROM FILE, SAVE PICTURE TO FILE.

CONVERT PICTURE

CONVERT PICTURE (picture; codec)

Parameter	Type		Description
picture	Picture	→	Picture to be converted
		←	Converted picture
codec	String	→	Picture Codec ID

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"), a Mime type (for example, "image/jpeg") or a 4-character QuickTime code (for example, "PNTG"). 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.

Example

Conversion of the *vpPhoto* picture to the jpeg format:

```
CONVERT PICTURE(vpPhoto;".jpg")
```

See Also

PICTURE CODEC LIST.

CREATE THUMBNAIL

version 6.7

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	→	Thumbnail colors in bits/pixels Default value = Current screen depth (0)

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.

Note: This command does not require QuickTime installation.

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.

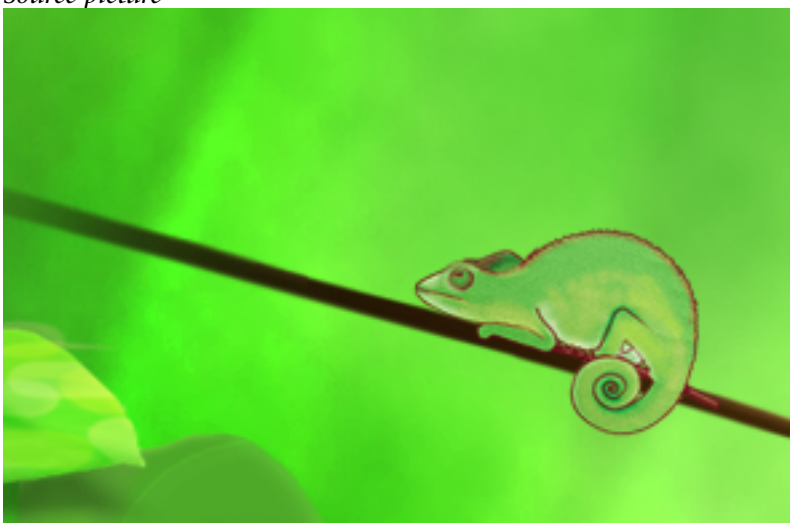
The optional parameter mode defines the thumbnail creation mode, i.e. the reduction mode. Three modes are available. The following predefined constants are provided by 4D in the "Picture Display Formats" constant theme:

Constants	Type	Value
Scaled to fit	Long integer	2
Scaled to fit proportional	Long integer	5
Scaled to fit prop centered	Long integer	6 (default)

Note: Only these constants can be used with CREATE THUMBNAIL. The other constants in the theme "Picture Display Formats" cannot be applied to this command.

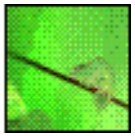
If you do not enter any parameter, the “Scaled to fit prop centered” mode (6) is applied by default. Below is an illustration of the various modes:

Source picture

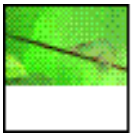


Resulting thumbnails (48x48)

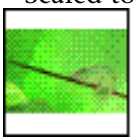
• Scaled to fit = 2



• 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 optional parameter depth defines the number of colors under Mac OS (i.e., the screen depth) to keep for the resulting thumbnail. The parameter is an integer equal to the number of bits per pixel: 1, 2, 4, 8, 16 or 32. Enter 0 to use the current screen depth (default value).

Note: Under Windows, the depth parameter is ignored; the command always uses the current screen depth.

GET PICTURE FROM LIBRARY (picRef | picName; picture)

Parameter	Type		Description
picRef | picName	Number | 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.

Note for components developers: If you want a 4D component to use graphics stored in the Picture Library, you must pass a picture name as first parameter. Indeed, when a component requiring its own pictures is installed by 4D Insider, the application can renumber automatically these new pictures if some database pictures have already the same reference number.

If there is no picture with that reference number or name, GET PICTURE FROM LIBRARY leaves picture unchanged.

Examples

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)
```

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)
```

3. See the third example for the command PICTURE LIBRARY LIST.

See Also

PICTURE LIBRARY LIST, REMOVE PICTURE FROM LIBRARY, SET PICTURE TO LIBRARY.

System Variables and Sets

If the Picture Library exists, the OK variable is set to 1. Otherwise, OK is set to zero.

Error Handling

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.

LOAD COMPRESS PICTURE FROM FILE

version 3

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	Number	→	Compression quality (1..1000)
picture	Picture	←	Compressed picture

Compatibility note: This command calls for obsolete mechanisms and is only kept for compatibility reasons. It has been favorably replaced by the READ PICTURE FILE and CONVERT PICTURE commands.

Description

This command compresses a picture loaded from a document on disk.

Note: This command does not work on Windows.

You can open a PICT document using the Open document function. You can then use the document reference returned by this function to load and compress the PICT found in the document. This command loads the picture into memory, compresses it using the method and quality you have specified, and then returns it into picture.

The picture is loaded into memory before it is compressed. If there is not enough memory to load the picture, use COMPRESS PICTURE FILE before calling LOAD COMPRESS PICTURE FROM FILE.

The parameter method is a 4-character string indicating the compressor type. You should pass one of the constants of the Picture compression theme in method. If method is an empty string, the picture is loaded but not compressed.

The parameter quality is an integer between 1 and 1000 indicating the quality of the compressed picture. In general, reducing the quality will allow for greater compression of the picture.

Warning: The compression ratio possible for a given quality depends on the size and nature of the picture you are compressing. Compressing small pictures may not produce any decrease in size.

Example

The following example presents an Open File dialog box that allows you to select a PICT file. The picture in the PICT file is loaded into memory, compressed, and stored in a picture variable. The file is then closed.

```
vRef:=Open document ("";"PICT")
If (OK=1)
    LOAD COMPRESS PICTURE FROM FILE(vRef;QT Photo compressor;500;vPict)
    CLOSE DOCUMENT(vRef)
End if
```

See Also

COMPRESS PICTURE, COMPRESS PICTURE FILE, CONVERT PICTURE, Pictures, READ PICTURE FILE, SAVE PICTURE TO FILE.

PICTURE CODEC LIST

version 11

PICTURE CODEC LIST (codecArray{; namesArray})

Parameter	Type		Description
codecArray	String array	←	IDs of available picture Codecs
namesArray	String array	←	Names of picture Codecx

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 both the Codec IDs of picture formats that are managed natively by 4D v11 (see below) as well as the IDs of any additional QuickTime Codecs that are installed on the machine (see the Pictures section).
These IDs can be passed in the format parameter for the WRITE PICTURE FILE and PICTURE TO BLOB commands that are used for exporting pictures

The Codec IDs can be returned in the codecArray array in three different forms:
• As an extension (for example, ".gif")
• As a Mime type (for example, "Picture/jpeg")
• As a 4-character QuickTime code (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.

See Also

PICTURE TYPE LIST, Pictures.

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.

Examples

1. The following code returns the catalog of the Picture Library in the arrays alPicRef and asPicName:

```
PICTURE LIBRARY LIST(alPicRef;asPicName)
```

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
```

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
                $vlPicture:=$vlNbPictures+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
```

See Also

GET PICTURE FROM LIBRARY, REMOVE PICTURE FROM LIBRARY, SET PICTURE TO LIBRARY.

PICTURE PROPERTIES (picture; width; height{; hOffset{; vOffset{; mode}}})

Parameter	Type		Description
picture	Picture	→	Picture for which to get information
width	Number	←	Width of the picture expressed in pixels
height	Number	←	Height of the picture expressed in pixels
hOffset	Number	←	Horizontal offset when displayed on background
vOffset	Number	←	Vertical offset when displayed on background
mode	Number	←	Transfer mode when displayed on background

Description

The PICTURE PROPERTIES command returns information about the picture you pass in picture.

The parameters width and height return the width and height of the picture.

The parameters hOffset, vOffset, and mode return the horizontal and vertical positions and the transfer mode of the picture when displayed on the background in a form (“On Background”).

See Also

Picture size.

Picture size

version 3

Picture size (picture) → Number

Parameter	Type		Description
picture	Picture	→	Picture for which to return the size in bytes
Function result	Number	←	Size in bytes of the picture

Description

This function returns the size of picture in bytes.

See Also

PICTURE PROPERTIES.

PICTURE TO BLOB

version 11 (Modified)

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"), a Mime type (for example "image/jpeg") or a 4-character QuickTime code (for example "PNTG"). 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 (QuickTime is not installed or the convertor is not available), OK is set to 0 and the generated BLOB is empty (0 byte).

See Also

BLOB TO PICTURE, PICTURE CODEC LIST, PICTURE TO GIF, PICTURE TYPE LIST, WRITE PICTURE FILE.

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 allows you to convert 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.

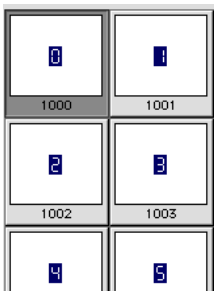
Note: The GIF picture format cannot contain more than 256 colors. If the original PICT picture contains more colors, some may be lost. The command reduces the number of colors according to the system palette. The GIF generated is of type 87a (opaque) and normal (not interlaced).

You can then save the picture located in blobGIF in a file using the BLOB TO DOCUMENT 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
            $blob:=gifcounter (<>nbHits)    `Calculates the GIF picture
                `The <>nbHits variable contains the number of connections
            SEND HTML BLOB ($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:

PICTURE TYPE LIST (formatArray{; nameArray})

Parameter	Type		Description
formatArray	String Array (4)	←	QuickTime codes for the available import/export formats
nameArray	String Array	←	Format names

Compatibility Note: This command has been kept for compatibility reasons. However, it requires QuickTime and does not provide access to formats managed natively by 4D starting with version 11. It is thus of limited interest and can be replaced favorably by the PICTURE CODEC LIST command.

Description

The PICTURE TYPE LIST command fills the formatArray array with picture import/export QuickTime codes available on the machine where it is executed.

The optional parameter nameArray array gets each picture format name. Format names are easier to understand than their codes.

QuickTime (version 4 minimum) needs to be installed on the machine where the command is executed. Otherwise, formatArray contains the PICT format only.

PICTURE TYPE LIST can be used to check that some picture formats are available for a given database. This command is useful when some specific formats, not installed by default, are necessary (a QuickTime 4 feature).
The information gathered in the nameArray array allow to build and to display a pop up menu containing the available picture export formats.

QuickTime 4 Conversion Codes

Below is the standard conversion code list provided by QuickTime 4. Each code is composed of 4 characters. Please note that as QuickTime 4 allows adding customized conversion routines, not all machines offer the same codes.

QuickTime 4 Codes	Names
PICT	QuickDraw PICT
PICS	PICS
GIFf	GIF
PNGf	PNG
TIFF	TIFF
8BPS	Photoshop (2.5 & 3.0)
SGI	Silicon Graphics
BMPf	BMP
JPEG	JPEG
JPEG	JFIF
PNTG	MacPaint
TPIC	TGA (Targa)
qdgx	QuickDraw GX Picture (if QuickDraw GX is installed)
qtif	QuickTime picture
FPix	FlashPix

See Also

BLOB TO PICTURE, PICTURE CODEC LIST, PICTURE TO BLOB, READ PICTURE FILE, WRITE 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	←	Picture from file

Description

The READ PICTURE FILE command allows you to open the picture saved in the fileName disk file and to load 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 just pass 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 is displayed and the user can select the file to be read, as well as the available formats.

You can obtain 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. In the case of custom formats using QuickTime, it is necessary to have QuickTime in order to read the picture within 4D.

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.

See Also

BLOB TO PICTURE, PICTURE CODEC LIST, PICTURE TYPE LIST, Pictures, WRITE PICTURE FILE.

REMOVE PICTURE FROM LIBRARY

version 6.7 (Modified)

REMOVE PICTURE FROM LIBRARY (picRef | picName)

Parameter	Type	Description
picRef | picName	Number | 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.

Examples

1. The following example deletes the picture #4444 from the Picture Library.

```
REMOVE PICTURE FROM LIBRARY(4444)
```

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
```

See Also

GET PICTURE FROM LIBRARY, PICTURE LIBRARY LIST, SET PICTURE TO LIBRARY.

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.

Description

This command saves picture in a document that was created using the Create document function.

Example

The following example creates a document and saves a picture in it:

```
vRef:=Create document("";"PICT")
If (OK=1)
    SAVE PICTURE TO FILE(vRef;vPict)
    CLOSE DOCUMENT(vRef)
End if
```

See Also

WRITE PICTURE FILE.

SET PICTURE TO LIBRARY

version 6.0.2

SET PICTURE TO LIBRARY (picture; picRef; picName)

Parameter	Type		Description
picture	Picture	→	New picture
picRef	Number	→	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: 31 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.

Examples

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")
```

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
```

See Also

GET PICTURE FROM LIBRARY, PICTURE LIBRARY LIST, REMOVE PICTURE FROM LIBRARY.

Error Handling

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 (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	Number	→	Transformation parameter
param2	Number	→	Transformation parameter
param3	Number	→	Transformation parameter
param4	Number	→	Transformation parameter

Description

The TRANSFORM PICTURE command can be used to apply 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 v11.

The source picture is modified directly after execution of the command. Note that except for “Scale” and “Fade to grey scale,” the operations are not destructive and can be reversed by carrying out the opposite operation or via 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:

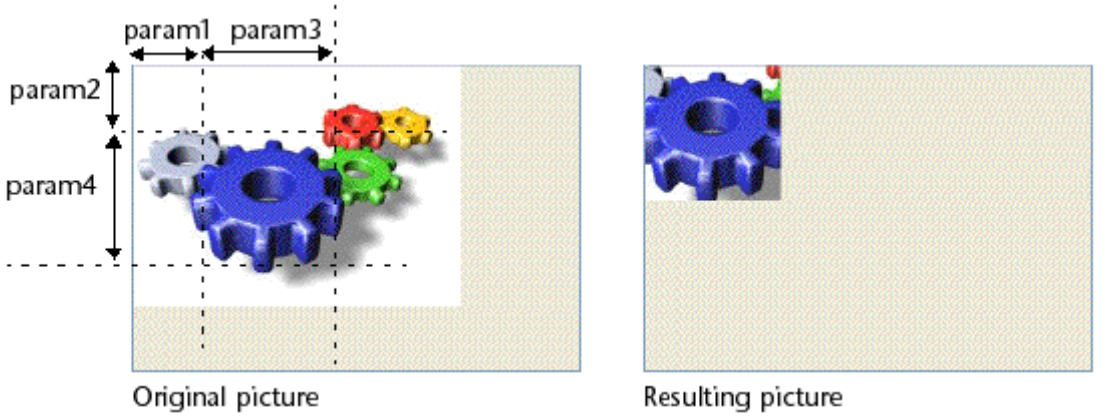
operator (value)	param1	param2	param3	param4	Values
Reset (0)	-	-	-	-	
Scale (1)	Width	Height	-	-	Factors
Translate (2)	X axis	Y axis	-	-	Pixels
Flip horizontally (3)	-	-	-	-	
Flip vertically (4)	-	-	-	-	
Crop (100)	X Orig.	Y Orig.	Width	Height	Pixels
Fade to grey scale (101)	-	-	-	-	

• 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.

Example

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)
```



See Also

COMBINE PICTURES, Picture Operators.

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 allows you to save 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. Under Windows, 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.

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"), a Mime type (for example "image/jpeg") or a 4-character QuickTime code (for example "PNTG"). You can get a list of available Codecs via the PICTURE CODEC LIST command.

If the format parameter is omitted or if QuickTime is not installed, the picture file is created with a PICT format.

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.

See Also

PICTURE TO BLOB, PICTURE TYPE LIST, Pictures, READ PICTURE FILE.

35

Printing

ACCUMULATE (data{; data2; ...; dataN})

Parameter	Type		Description
data	Field or 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 BREAK LEVEL 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.

See Also

BREAK LEVEL, ORDER BY, PRINT SELECTION, Subtotal.

BREAK LEVEL (level{; pageBreak})

Parameter	Type		Description
level	Number	→	Number of break levels
pageBreak	Number	→	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
OUTPUT FORM ([Emp];"Dept salary")  ` Select the report form
PRINT SELECTION([Emp])  ` Print the report
```

See Also

ACCUMULATE, ORDER BY, PRINT SELECTION, Subtotal.

CLOSE PRINTING JOB

Parameter	Type	Description
This command does not require any parameters		

Description

The CLOSE PRINTING JOB command can be used to close the print job previously opened by the OPEN PRINTING JOB command and to send 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.

See Also

OPEN PRINTING JOB.

Get current printer → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Name of the current printer

Description

Note: This command does not work under Mac OS 9. Under Windows, it requires at least Windows 2000.

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.

See Also

PRINTERS LIST, SET CURRENT PRINTER.

System Variables or Sets

If no printer is installed, the system variable *OK* is set to 0. Otherwise, it is set to 1.

Get print marker (markNum) → Number

Parameter	Type		Description
markNum	Number	→	Marker number
Function result	Number	←	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 Header	Longint	200
Form Header1...10	Longint	201...210
Form Detail	Longint	0
Form Break0...9	Longint	300...309
Form Footer	Longint	100

Example

Refer to the example of the SET PRINT MARKER command.

See Also

MOVE OBJECT, SET PRINT MARKER.

GET PRINT OPTION (option; value1{; value2})

Parameter	Type		Description
option	Longint	→	Option number
value1	Longint | String	←	Value 1 of the option
value2	Longint	←	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 pass a value or one of the following predefined constants, located in the "Print options" theme:

Constant	Type	Value
Paper option	Longint	1
Orientation option	Longint	2
Scale option	Longint	3
Number of copies option	Longint	4
Paper source option	Longint	5
Color option	Longint	8
Destination option	Longint	9
Double sided option	Longint	11
Spooler document name option	Longint	12
Mac spool file format option	Longint	13
Hide printing progress option	Longint	14

The command returns, in the value1 and (optionally) value2 parameters, the current value(s) of the specified option. For more information on options and possible values, refer to the description of the SET PRINT OPTION command. Note the following specific features of the GET PRINT OPTION command:

• option = 1 (paper option): returns the name of the current paper in value1 if value2 is omitted. If value2 is passed, the command returns, respectively, the width and height of the paper in value1 and value2. Use the PRINT OPTION VALUES command to get the name, height and width of all the paper formats offered by the printer.

• option = 2 (orientation option): returns 1 (Portrait) or 2 (Landscape). If a different orientation option is used, value1 is set to 0.

• option = 5 (paper source option): in value1, returns the index (in the array of trays returned by the PRINT OPTION VALUES command) of the paper tray used (value2 must be omitted).

Note: This option can only be used under Windows.

• option = 8 (color option): returns a code in value1 specifying the mode for handling color: 1=Black and white (monochrome), 2=Color.
Note: This option can only be used under Windows.
• option = 9 (destination 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. If value1 contains a predefined value different from 1 or 5, value2 contains the access path of the printed file.
• option = 11 (double sided option): returns 0 (Standard or Single-sided, default value) or 1 (Double-sided) in value1. If value1 equals 1, value2 may return one of the following values: 0=Left binding (default), 1=Top binding.
Note: This option can only be used under Windows.
• option = 12 (spooler document name option): returns the name of the current print document in value1, if it has been defined previously. Otherwise, an empty string is returned.

Note: The GET PRINT OPTION command only operates with PostScript printers.

See Also

PRINT OPTION VALUES, SET PRINT OPTION.

System Variables or Sets

The system variable *OK* is set to 1 if the command has been executed correctly; otherwise, it is set to 0.

Constants

Print options theme.

GET PRINTABLE AREA (height{; width})

Parameter	Type		Description
height	Number	←	Height of printable area
width	Number	←	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
```

See Also

GET PRINTABLE MARGIN, Print form.

GET PRINTABLE MARGIN (left; top; right; bottom)

Parameter	Type		Description
left	Number	←	Left margin
top	Number	←	Top margin
right	Number	←	Right margin
bottom	Number	←	Bottom margin

Description

The GET PRINTABLE MARGIN command returns the current values of the different margins defined using the Print form command.

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

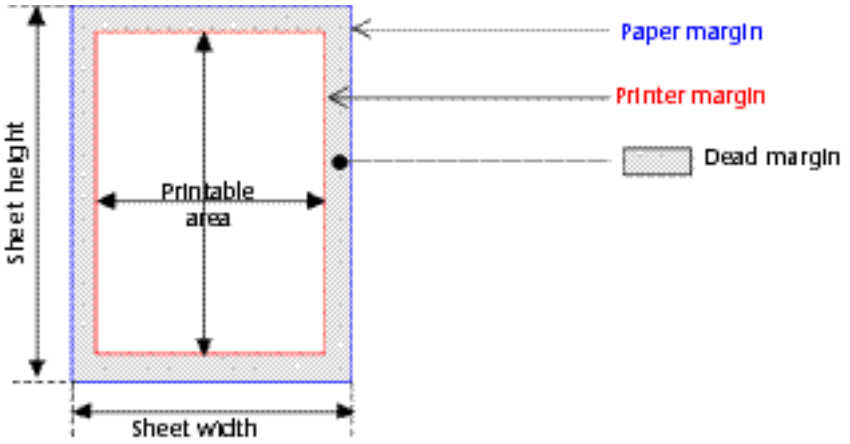
By default, the printing calculation in 4D is based on “printable margins”. The advantage of this system is that the forms adapt themselves automatically to the new printers (since they are positioned in the printable area). On the other hand, in the case of pre-printed forms, it was not possible to position the elements to be printed precisely because changing the printer could modify the printable margins.

Beginning with 4D version 6.8.1, it is possible to base the form printing carried out using the Print form, PRINT RECORD and PRINT SELECTION 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 referes to the area located between the paper margin and the printer margin.



See Also

GET PRINTABLE AREA, Print form, SET PRINTABLE MARGIN.

Get printed height

version 6.8.1

Get printed height → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	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.

See Also

GET PRINTABLE AREA, Print form, SET PRINTABLE MARGIN.

Level → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	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 Details)
            ` 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
```

See Also

ACCUMULATE, BREAK LEVEL, Form event, PRINT SELECTION.

OPEN PRINTING JOB

Parameter	Type	Description

This command 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 and 4D Draw plug-ins, 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. It is possible to open as many print jobs as there are processes. Naturally, in this case it is necessary to have several printers since each printer will be busy until the end of the job.

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.

See Also

CLOSE PRINTING JOB.

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 systemvariable *OK* to 0.

Examples

1. See example for the Print form command.

2. Refer to the example of the SET PRINT MARKER command.

See Also

CANCEL, Print form.

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)
```

See Also

Print form, PRINT RECORD, PRINT SELECTION, SET PRINT OPTION.

Print form ({aTable; }form{; area1{; area2}}){ → Number }

Parameter	Type		Description
aTable	Table	→	Table owning the form, or Default table, if omitted
form	String	→	Form to print
area1	Number	→	Print marker, or Beginning area (if area2 is specified)
area2	Number	→	Ending area (if area1 specified)
Function result	Number	←	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 Header	Longint	200
Form Header1...10	Longint	201...210
Form Detail	Longint	0
Form Break0...9	Longint	300...309
Form Footer	Longint	100

• 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. Otherwise, if the last page is not full, it stays in memory and is not printed.

Starting with version 2004.5 of 4D, 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.

Examples

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
```

2. Refer to the example of the SET PRINT MARKER command.

See Also

CANCEL, PAGE BREAK, PAGE SETUP, PRINT SETTINGS.

PRINT LABEL ({aTable}{; document{; * | >}})

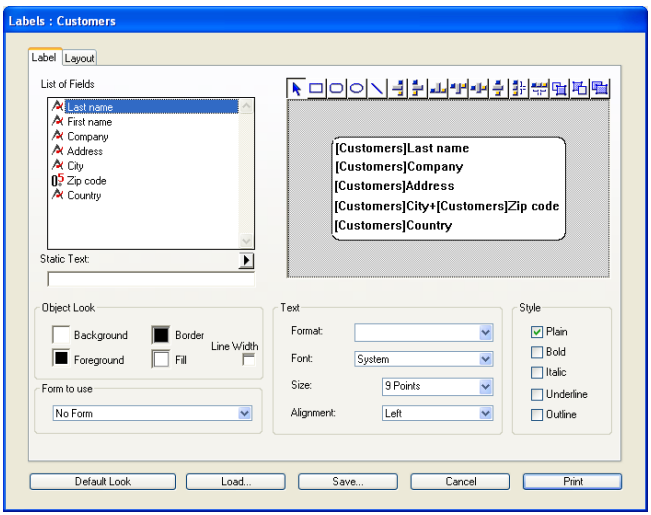
Parameter	Type		Description
aTable	Table	→	Table to print, or Default table, if omitted
document	String	→	Name of disk label document
* | >		→	* to suppress the printing dialog boxes, or > to not reinitialize print settings

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.



By default, PRINT LABEL displays the printer dialog boxes 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 (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 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.

Examples

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
OUTPUT FORM ([Addresses]; "Label Out") ` Select the output form
PRINT LABEL([Addresses]) ` Print the labels
OUTPUT FORM ([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
```

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
```

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
```

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
```

See Also

PRINT SELECTION, QR REPORT.

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
Paper option	Longint	1
Paper source option	Longint	5

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.

See Also

GET PRINT OPTION, SET PRINT OPTION.

Constants

Print options theme.

PRINT RECORD ({aTable}{; }{* | >})

Parameter	Type		Description
aTable	Table	→	Table for which to print the current record or Default table if omitted
* | >	* | >	→	* 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 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.

Examples

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.

```
    ` Select the right output form for printing
OUTPUT FORM([Invoices];"Print One From Data Entry")
    ` Print Invoices as it is (without showing the printing dialog boxes)
PRINT RECORD([Invoices];*)
OUTPUT FORM([Invoices];"Standard Output") ` Restore the previous output form
```

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)
    OUTPUT FORM([Employees];"Detailed")  `Use the first print form
    PRINT RECORD([Employees];>)  `Print using user-defined parameters
    OUTPUT FORM([Employees];"Simple")  `Use the second print form
    PRINT RECORD([Employees];>)  `Print using user-defined parameters
    OUTPUT FORM([Employees];"Output")  `Restore default output form
End if
```

See Also

Print form.

PRINT SELECTION ({aTable}{; }{* | >})

Parameter	Type		Description
aTable	Table	→	Table for which to print the selection, or Default table, if omitted
* | >	* | >	→	* 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 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
```

See Also

ACCUMULATE, BREAK LEVEL, Level, PAGE SETUP, Subtotal.

PRINT SETTINGS {(dialType)}

Parameter	Type		Description
dialType	Longint	→	Dialog box(es) to be displayed: 0 (or parameter omitted) = All 1 = Print Setup, 2 = Print Job

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:
• If you pass 0 in dialType or omit this parameter, both printing dialog boxes are displayed. First, it displays the Print Setup dialog box. Then, it displays the Print Job dialog box.
• If you pass 1 in dialType, only the Print Setup dialog box is displayed. The current printing options will be used.
• If you pass 2 in dialType, only the Print Job dialog box is displayed. The current print settings will be used.

Note: The Print Job dialog box contains a Preview on Screen check box that allows the user to specify to print to the screen. You can preset or reset this check bok by calling SET PRINT PREVIEW before calling PRINT SETTINGS.

Example

See example for the command PRINT FORM.

System Variables or 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.

See Also

OPEN PRINTING JOB, PAGE BREAK, Print form, SET PRINT PREVIEW.

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 (Windows only)

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. This parameter can only be used under Windows.

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.

See Also

Get current printer, SET CURRENT PRINTER.

System Variables or 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 → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	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
```

See Also

PRINT SELECTION.

SET CURRENT PRINTER (printerName)

Parameter	Type		Description
printerName	String	→	Name of printer to be used

Description

Note: This command does not work under Mac OS 9. Under Windows, it requires at least Windows 2000.

The SET CURRENT PRINTER command is used to designate 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 new PRINTERS LIST command beforehand.
If you pass an empty string in printerName, the current printer defined in the system will be used.

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 LABEL, 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.

See Also

Get current printer, PRINTERS LIST.

System Variables or 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.

SET PRINT MARKER (markNum; position{; *})

Parameter	Type		Description
markNum	Number	→	Marker number
position	Number	→	New position for the marker
*	*	→	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, MOVE OBJECT 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.

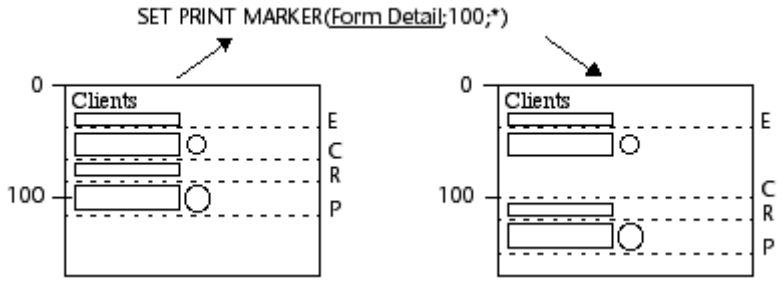
Pass one of the constants of the Form area theme in the markNum parameter:

Constant	Type	Value
Form Header	Longint	200
Form Header1...10	Longint	201...210
Form Detail	Longint	0
Form Break0...9	Longint	300...309
Form Footer	Longint	100

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.



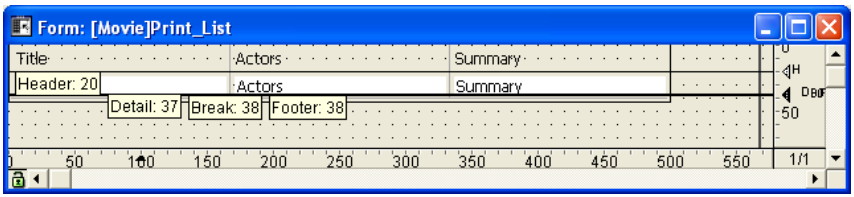
Notes:

• 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 G	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 Huff, 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
      GET OBJECT RECT([Film]Actors;$l;$t;$r;$b)
      $fixed_wdth:=$r-$l   `Calculation of the Actors text field size
      $exact_hght:=$b-$t
      BEST OBJECT SIZE([Film]Actors;$wdth;$hght;$fixed_wdth)
         `Optimal size of the field according to its contents
      $movement:=$hght-$exact_hght

      GET OBJECT RECT([Film]Summary;$l1;$t1;$r1;$b1)
      $fixed_wdth1:=$r1-$l1   `Calculation of the Summary text field size
      $exact_hght1:=$b1-$t1
      BEST OBJECT SIZE([Film]Summary;$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
         MOVE OBJECT([Film]Actors;$l;$t;$r;$hght+$t;*)
         MOVE OBJECT([Film]Summary;$l1;$t1;$r1;$hght1+$t1;*)

            `Resizing of dividing lines
         GET OBJECT RECT(*;"H1Line";$l;$t;$r;$b)
         MOVE OBJECT(*;"H1Line";$l;$final_pos-1;$r;$final_pos;*)
         For ($i;1;4;1)
            GET OBJECT RECT(*;"VLine"+String($i);$l;$t;$r;$b)
            MOVE OBJECT(*;"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
```

See Also

BEST OBJECT SIZE, GET OBJECT RECT, Get print marker, MOVE OBJECT, PAGE BREAK, Print form, PRINT RECORD, PRINT SELECTION.

SET PRINT OPTION (option; value1{; value2})

Parameter	Type		Description
option	Longint	→	Option number
value1	Longint | String	→	Value 1 of the option
value2	Longint | String	→	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.

The option parameter allows you to indicate the option to be modified. You can pass either a value or one of the predefined constants of the "Print options" theme.
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.

The following table lists the options and their possible values:

option (constant)	value1	value2
1 (Paper option)	Name	-
	Width	Height
2 (Orientation option)	1=Portrait, 2=Landscape	-
3 (Scale option)	Number (%)	-
4 (Number of copies option)	Number	-
5 (Paper source option)	*Windows only*:	
	Index (number)	-
8 (Color option)	*Windows only*:	
	1=N/B, 2=Color	-
9 (Destination option)	1=Printer,	-
	2=File (PC)/PS (Mac),	Access path
	3=PDF (Mac),	Access path
	5=Screen (Mac)	-
11 (Double sided option)	*Windows only*:	
	0=Single-sided (standard)	-
	1=Double-sided	Binding: 0=Left (default), 1=Top

12 (Spooler document name option)	Name of document to be printed	-
13 (Mac spool file format option)	0=PDF mode, 1= PostScript mode	-
14 (Hide printing progress option)	0=Display (default), 1=Hide	-

• Paper option (1): the list of all the names of available paper can be obtained using the PRINT OPTION VALUES command.
You can either pass the name of the paper in value1 (and, in this case, omit value2), or pass the paper width in value1 and its height in value2. The width and height must be expressed in screen pixels.
• Orientation option (2): you can pass either 1 (Portrait), or 2 (Landscape) in value1.
• Scale option (3): pass a percentage in value1. 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 (4): pass the number of copies to be printed in value1.
• Paper source option (5): pass the number corresponding to the index, in the array of trays returned by the PRINT OPTION VALUES command, of the paper tray to be used.
Note: This option can only be used under Windows.
• Color option (8): in value1, pass the code specifying the mode for handling color: 1=Black and white (monochrome), 2=Color.
Note: This option can only be used under Windows.
• Destination option (9): in value1, pass the code specifying the type of print destination: 1=Printer, 2=File (PC)/PS (Mac), 3=PDF file (Mac OS only), 5=Screen (Mac OS X driver option).
If value1 is different from 1 or 5, pass the pathname for the resulting document in value2. 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. Under Windows only: if you pass an empty string in value2 or omit this parameter, a file saving dialog appears at the time of printing.
• Double sided option (11): you can either pass 0 (Single-sided or standard), or 1 (Double-sided) in value1. If value1 equals 1, you can define the binding to be applied using value2: 0=Left binding (default value), 1=Top binding.
Note: This option can only be used under Windows.
• Spooler document name option (12): in value1, pass the name of the print document that must appear in the list of spooler documents.
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.
Warning: 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.

• Mac spool file format option (13): in value1, pass 0 to set the print job in PDF mode (default value) and 1 to "force" the print job in PostScript mode. This option has no effect under Windows.
Note: Under Mac 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 Mac OS X for the current session. Keep in mind that printing in PostScript mode can lead to undesired side effects.
• Hide printing progress option (14): pass 1 in value1 to hide the progress windows and 0 to display them again (default operation). This option is particularly useful in the case of PDF printing under Mac OS X.
Note: There is already a Printing progress option found in the Preferences dialog box (Application/Options page). However, it is applied globally to the application and does not hide all the windows under Mac OS X.

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 LABEL, PRINT RECORD, Print form, and QR REPORT 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 LABEL, 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 only operates with PostScript printers.

See Also

GET PRINT OPTION, PRINT OPTION VALUES, SET CURRENT PRINTER.

System Variables or Sets

The system variable *OK* is set to 1 if the command has been executed correctly; otherwise, it is set to 0.

Error Handling

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.

Constants

Print options theme.

SET PRINT PREVIEW

version 3

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
```

See Also

PRINT RECORD, PRINT SELECTION, PRINT SETTINGS.

SET PRINTABLE MARGIN (left; top; right; bottom)

Parameter	Type		Description
left	Number	→	Left margin
top	Number	→	Top margin
right	Number	→	Right margin
bottom	Number	→	Bottom margin

Description

The SET PRINTABLE MARGIN command enables you to set the values of various printing margins by using the Print form command.

You can pass one of the following values into 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.

Examples

1. The following example enables you to obtain 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
```

2. The following example enables you to obtain 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
```

See Also

GET PRINTABLE MARGIN, Get printed height, Print form.

Subtotal (data{; pageBreak}) → Number

Parameter	Type		Description
data	Field	→	Numeric field or variable to return subtotal
pageBreak	Number	→	Break level for which to cause a page break
Function result	Number	←	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 ACCUMULATE and BREAK LEVEL 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.

See Also

ACCUMULATE, BREAK LEVEL, Level, PRINT SELECTION.

36

Process (Communications)

CALL PROCESS (process)

Parameter	Type		Description
process	Number	→	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 commands GET PROCESS VARIABLE and SET PROCESS VARIABLE.

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.

See Also

Form event, GET PROCESS VARIABLE, SET PROCESS VARIABLE.

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.

See Also

Semaphore.

GET PROCESS VARIABLE (process; srcVar; dstVar{; srcVar2; dstVar2; ...; srcVarN; dstVarN})

Parameter	Type		Description
process	Number	→	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.

Examples

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)
```

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)
```

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.

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.

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
```

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)
```

7. See the example for the command DRAG AND DROP PROPERTIES.

See Also

CALL PROCESS, Drag and Drop, DRAG AND DROP PROPERTIES, Processes, SET PROCESS VARIABLE, VARIABLE TO VARIABLE.

Semaphore (semaphore{; tickCount}) → Boolean

Parameter	Type		Description
semaphore	String	→	Semaphore to test and set
tickCount	Integer	→	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 (each user's computer) 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. By creating and testing semaphores, methods can communicate between workstations.

The Semaphore function returns TRUE if semaphore 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. semaphore is limited to 255 characters, including prefixes (<>, $). If you pass a longer string, the semaphore will be tested with the truncated string.

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 4D Server, global semaphores are shared among all the processes running on all clients. A local semaphore is only shared among the processes running on the client where it has been created.
In 4D Developer, 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.

You do not use semaphores to protect record access. This is automatically done by 4D Developer and 4D Server. Use semaphores to prevent several users from performing the same operation at the same time.

Examples

1. 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
```

2. 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.

See Also

CLEAR SEMAPHORE, Test semaphore.

SET PROCESS VARIABLE (process; dstVar; expr{; dstVar2; expr2; ...; dstVarN; exprN})

Parameter	Type		Description
process	Number	→	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 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, 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.

Examples

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;"")
```

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)
```

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.

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.

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)
```

See Also

CALL PROCESS, GET PROCESS VARIABLE, Processes, VARIABLE TO VARIABLE.

Test semaphore (semaphore) → Boolean

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 allows you to test 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 exit. 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
```

See Also

Semaphore.

VARIABLE TO VARIABLE (process; dstVar; srcVar{; dstVar2; srcVar2; ...; dstVarN; srcVarN})

Parameter	Type		Description
process	Number	→	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)
```

See Also

GET PROCESS VARIABLE, Processes, SET PROCESS VARIABLE.

37

Process (User Interface)

BRING TO FRONT (process)

Parameter	Type		Description
process	Number	→	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. The order of the windows is retained. 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.

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
```

See Also

HIDE PROCESS, Process state, SHOW PROCESS.

Frontmost process {(*)} → Integer

Parameter	Type		Description
*		→	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.

See Also

BRING TO FRONT, WINDOW LIST.

HIDE PROCESS (process)

Parameter	Type		Description
process	Number	→	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)
```

See Also

Process state, SHOW PROCESS.

SHOW PROCESS (process)

Parameter	Type		Description
process	Number	→	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)
```

See Also

BRING TO FRONT, HIDE PROCESS, Process state.

38

Processes

version 11 (Modified)

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 section Stored Procedures 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.

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.

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.

Note: 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. UserSet and LockedSet are process sets. 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 section Web server configuration and connection management 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 Developer 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.

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.

See Also

Methods, Project Methods, Variables.

Count tasks

Count tasks → Integer

Parameter	Type		Description
This command does not require any parameters			
Function result	Integer	←	Number of open processes (including kernel processes)

Description

Count tasks returns the number of processes open in 4D Client, 4D Server (stored procedures) or in 4D Developer.

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 On Exit Database Method.

See Also

Count user processes, Count users, PROCESS PROPERTIES, Process state.

Count user processes

version 11 (Modified)

Count user processes → Integer

Parameter	Type		Description
This command does not require any parameters			
Function result	Integer	←	Number of open processes (excluding kernel processes)

Description

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).

See Also

Count tasks, Count users.

Count users

version 3

Count users → Integer

Parameter	Type		Description
This command does not require any parameters			
Function result	Integer	←	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 4D Developer, Count users returns 1.

See Also

Count tasks, Count user processes.

Current process

Current process → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	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.

See Also

Process number, PROCESS PROPERTIES, Process state.

DELAY PROCESS

DELAY PROCESS (process; duration)

Parameter	Type		Description
process	Number	→	Process number
duration	Number	→	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.

If the process is already delayed, this command delays it again. The parameter duration 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).

Examples

1. See example in Record Locking.
2. See example for the command Process number.

See Also

HIDE PROCESS, PAUSE PROCESS.

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.

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.

Examples

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")
```

2. If you want all the clients to execute the "EmptyTemp" method:

```
EXECUTE ON CLIENT("@";"EmptyTemp")
```

3. Refer to the example of the REGISTER CLIENT command.

See Also

GET REGISTERED CLIENTS, REGISTER CLIENT, UNREGISTER CLIENT.

Execute on server (procedure; stack{; name{; param{; param2; ...; paramN}{; *}}}) → Number

Parameter	Type		Description
procedure	String	→	Procedure to be executed within the process
stack	Number	→	Stack size in bytes
name	String	→	Name of the process created
param	Expression	→	Parameter(s) to the procedure
*		→	Unique process
Function result	Number	←	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 Execute on server 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 Execute on server on the Server machine, Execute on server 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 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: In stack, you pass 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. It is expressed in bytes; it is recommended to pass at least 64K (around 64,000 bytes), but you can pass more if the process can perform large chain calls (subroutines calling subroutines in cascade). For example, you can pass 200K (around 200000 bytes), if necessary.

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 only holds local variables, method calls, parameters in subroutines and stacked records.

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 command PROCESS PROPERTIES 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.

A process name can be up to 31 characters long. 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: Starting with version 6, 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.

The 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)
    INPUT FORM([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_STRING(31;$2)
C_TIME($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";32*1024;
                                        "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_STRING(31;$2)
C_BLOB($3)
C_LONGINT(spErrCode)

    ` Operation is not finished yet, set spErrCode to 1
spErrCode:=1
$vpTable:=Table($1)
INPUT FORM($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.

See Also

New process, Stored Procedures.

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).

Note: If the operation was successful, the OK system variable is equal to 1.

Examples

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)
```

2. Refer to the example of the REGISTER CLIENT command.

See Also

EXECUTE ON CLIENT, REGISTER CLIENT, UNREGISTER CLIENT.

New process

version 2004.3 (Modified)

New process (method; stack{; name{; param{; param2; ...; paramN}{; *}}}) → Number

Parameter	Type		Description
method	String	→	Method to be executed within the process
stack	Number	→	Stack size in bytes
name	String	→	Name of the process created
param	Expression	→	Parameter(s) to the method
*		→	Unique process
Function result	Number	←	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: In stack, you pass 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. It is expressed in bytes; it is recommended to pass at least 64K (around 64,000 bytes), but you can pass more if the process can perform large chain calls (subroutines calling subroutines in cascade). For example, you can pass 200K (around 200000 bytes), if necesary.

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 4D informations ; the amount of information kept on the stack depends on the number of nested methods 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.

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 command PROCESS PROPERTIES when applied to this new process. A process name can be up to 31 characters long. 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.
Parameter to Process Method: Starting with version 6, 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.

The 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";32*1024;"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
$vlProcessID:=New process("Add Customers";64*1024;"Adding Customers";*)
If ($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.

See Also

Execute on server, Methods, Processes, Project Methods, Variables.

PAUSE PROCESS

PAUSE PROCESS (process)

Parameter	Type		Description
process	Number	→	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. PAUSE PROCESS 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).

See Also

DELAY PROCESS, HIDE PROCESS, RESUME PROCESS.

Process aborted → Boolean

Parameter	Type		Description
This command does not require any parameters			
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.

Example

This command can be used as a particular type of programming on the Web server, only in compiled mode. When you use a method to send Web pages by using a loop like While...End while (see example), the mechanism of the Web server doesn't allow you to stop the loop in case of a timeout (end of the inactivity period authorized) on a Web browser. If the Web process is not closed, a context is therefore still in use.
The Process aborted command, placed in the initial test of the loop, will return True in case of a timeout. The loop can then be interrupted and the process can be aborted.
Here is a method that can be used to send HTML pages. In compiled mode, this loop cannot be interrupted in case of a timeout:

```
While (True)
    SEND HTML FILE (HTMLFile)
End while
```

The Process aborted command allows you to use the same type of method, while still being able to exit the loop and abort the Web process in case of a timeout:

```
While (Not (Process aborted))
    SEND HTML FILE (HTMLFile)
End while
```

Process number (name{; *}) → Number

Parameter	Type		Description
name	String	→	Name of process for which to retrieve the process number
*		→	Return the process number from the server
Function result	Number	←	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 and SET PROCESS 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
```

See Also

GET PROCESS VARIABLE, PROCESS PROPERTIES, Process state, SET PROCESS VARIABLE.

PROCESS PROPERTIES

version 11 (Modified)

PROCESS PROPERTIES (process; procName; procState; procTime{; procVisible{; uniqueID{; origin}}})

Parameter	Type		Description
process	Number	→	Process number
procName	String	←	Process name
procState	Number	←	Process state
procTime	Number	←	Cumulative time taken by process in ticks
procVisible	Boolean	←	Visible (TRUE) or Hidden (FALSE)
uniqueID	Integer	←	Unique process number
origin	Longint	←	Origin of the process

Description

The PROCESS PROPERTIES command returns 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 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	Long Integer	-1
Delayed	Long Integer	1
Does not exist	Long Integer	-100
Executing	Long Integer	0
Hidden modal dialog	Long Integer	6
Paused	Long Integer	5

Waiting for input output	Long Integer	3
Waiting for internal flag	Long Integer	4
Waiting for user event	Long Integer	2

• procTime returns the cumulative time that the process has used since it started, in ticks (1/60th of a second) .

• procVisible, if specified, returns TRUE if the process is visible, FALSE if hidden.

• 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
Monitor Process	Longint	-26
Server Controller Process	Longint	-25
SQL Method Execution Process	Longint	-24
Timer Process	Longint	-23
MSC Process	Longint	-22
Restore Process	Longint	-21
Log File Process	Longint	-20
Backup Process	Longint	-19
Internal 4D Server Process	Longint	-18
Method editor macro Process	Longint	-17
On Quit Process	Longint	-16
4D Server Process	Longint	-15
Execute on Client Process	Longint	-14
Web server Process	Longint	-13
Web Process on 4D Client	Longint	-12
Web Process with Context	Longint	-11
Other 4D Process	Longint	-10
External Task	Longint	-9
Event Manager	Longint	-8
Apple Event Manager	Longint	-7
Serial Port Manager	Longint	-6
Indexing Process	Longint	-5
Cache Manager	Longint	-4

Web Process with no Context	Longint	-3
Design Process	Longint	-2
Main Process	Longint	-1
None	Longint	0
Execute on Server Process	Longint	1
Created from Menu Command	Longint	2
Created from execution dialog	Longint	3
Other User Process	Longint	4

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.

Examples

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)
```

2. See example for On Exit Database Method.

See Also

Count tasks, Process state.

Process state (process) → Number

Parameter	Type		Description
process	Number	→	Process number
Function result	Number	←	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	Long Integer	-1
Delayed	Long Integer	1
Does not exist	Long Integer	-100
Executing	Long Integer	0
Hidden modal dialog	Long Integer	6
Paused	Long Integer	5
Waiting for input output	Long Integer	3
Waiting for internal flag	Long Integer	4
Waiting for user event	Long Integer	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 STRING(31;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 STRING(31;asProcName;$vlActualCount)
ARRAY INTEGER(aiProcNum;$vlActualCount)
```

See Also

Count tasks, PROCESS PROPERTIES.

REGISTER CLIENT (clientName{; period{; *}})

Parameter	Type		Description
clientName	String	→	Name of the 4D Client session
period	Longint	→	Server's interrogation period (in seconds)
*	*	→	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.

Note: 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.

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.

After executing the command, the client station will periodically ask 4D Server to see if another 4D Client or the server itself is calling it.
By default, this interrogation is done every two seconds. You can modify this time period by modifying period. The minimum value is one second.

Note: If this command is used with 4D Developer, it has no effect.

Once the command is executed, it is not possible to modify 4D Client's name or the server's interrogation period on the fly. To do so, you must call the UNREGISTER CLIENT command, then the REGISTER CLIENT command.

Note: More than one 4D Client can have the same registered name.

If 4D Client is correctly registered, the OK system variable is equal to 1. If 4D Client was already registered, the command doesn't do anything and OK is equal to 0.

Examples

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 4D 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 Client)
      ` 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
```

See Also

EXECUTE ON CLIENT, GET REGISTERED CLIENTS, UNREGISTER CLIENT.

System Variables and Sets

If 4D Client is correctly registered, the OK system variable is equal to 1. If 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	Number	→	Process number

Description

RESUME PROCESS resumes a process whose execution has been paused or delayed. If process is not paused or delayed, RESUME PROCESS 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).

See Also

DELAY PROCESS, PAUSE PROCESS.

UNREGISTER CLIENT

Parameter	Type	Description
This command 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 Developer, the command has no effect.

If the client is correctly unregistered, the OK system variable is equal to 1. If the client wasn't registered, OK is equal to 0.

Example

Refer to the example for the REGISTER CLIENT command.

See Also

EXECUTE ON CLIENT, GET REGISTERED CLIENTS, REGISTER CLIENT.

39

Queries

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 can be used to enable or disable the query analysis mode for the current process. The command takes into account both queries carried out via the 4D language or by SQL.

Callingthe 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 new 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 in the case of an SQL query:

```
DESCRIBE QUERY EXECUTION(True) `analysis mode
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
Begin SQL
   SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName),
                                                          CITIES.City_Name
      FROM ACTORS, CITIES
      WHERE ACTORS.Birth_City_ID=CITIES.City_ID
      ORDER BY 2,1
      INTO :aTitles, :aDirectors;
End SQL
vResultPlan:=Get Last Query Plan(Description in Text Format)
vResultPath:=Get Last Query Path(Description in Text Format)
DESCRIBE QUERY EXECUTION(False)  `End analysis mode
```

After this code is executed, vResultPlan and vResultPath contain descriptions of the queries carried out, for example:

```
vResultPlan:
      [Join] : ACTORS.Birth_City_ID = CITIES.City_ID
vResultPath:
      And
         [Merge] : ACTORS with CITIES
            [Join] : ACTORS.Birth_City_ID = CITIES.City_ID  (227 records found in 0 ms)
         → 227 records found in 0 ms
      → 227 records found in 0 ms
```

See Also

Get Last Query Path, Get Last Query Plan.

Find in field

version 11 (Modified)

Find in field (targetField; value) → Longint

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.

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.

Example

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
```

Get Last Query Path

version 11.1

Get Last Query Path (descFormat) → String

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.

See Also

DESCRIBE QUERY EXECUTION, Get Last Query Plan.

Get Last Query Plan

version 11.1

Get Last Query Plan (descFormat) → String

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.

See Also

DESCRIBE QUERY EXECUTION, Get Last Query Path.

ORDER BY ({aTable}{; aField{; > or <{; aField2; > or <2; ...; aFieldN; > or <N}{; *}}})

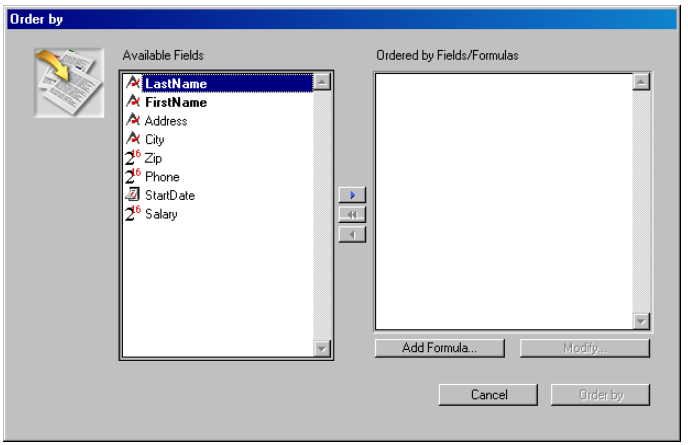
Parameter	Type		Description
aTable	Table	→	Table for which to order selected records, or Default table, if omitted
aField	Field	→	Field on which to set the order for each level
> or <		→	Ordering direction for each level: > to order in ascending order, or < to order in descending order
*		→	Continue 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 no default table has been set, an error occurs.

If you do not specify the aField, the > or < or the * parameters, ORDER BY displays the Order By editor for table. The Order By editor is shown here:



For more information about using the Order By editor, refer to the *4D Design Reference* manual.

The user builds the sort, then clicks the Sort button to perform the sort. If the sort is performed without interruption, the OK variable is set to 1. If the user clicks Cancel, the ORDER BY terminates with no sort actually performed, and sets the OK variable to 0 (zero).

Examples

1. The following line displays the Order By editor for the [Products] table:

```
ORDER BY([Products])
```

2. The following line displays the Order By editor for the default table (if it has been set):

```
ORDER BY
```

If you specify the field 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 field 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.

Examples

3. The following line orders the selection of [Products] by name in ascending order:

```
ORDER BY([Products];[Products]Name;>)
```

4. The following line orders the selection of [Products] by name in descending order:

```
ORDER BY([Products];[Products]Name;<)
```

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;>)
```

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;<)
```

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;<)
```

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;>)
```

If you omit the sorting order parameter > or <, ascending order is the default.

Example

9. The following line orders the selection of [Products] by name in ascending order:

```
ORDER BY([Products];[Products]Name)
```

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. The field may belong to the (selection's) table being reordered or to a One table related to table with an automatic relation. (Remember, the table to which ORDER BY is applied must be the Many table.) In this case, the sort is always sequential.

Examples

10. The following line performs an indexed sort if [Products]Name is indexed:

```
ORDER BY([Products];[Products]Name;>)
```

11. The following line performs a sequential sort, whether or not the fields are indexed:

```
ORDER BY([Products];[Products]Type;>;[Products]Price;>)
```

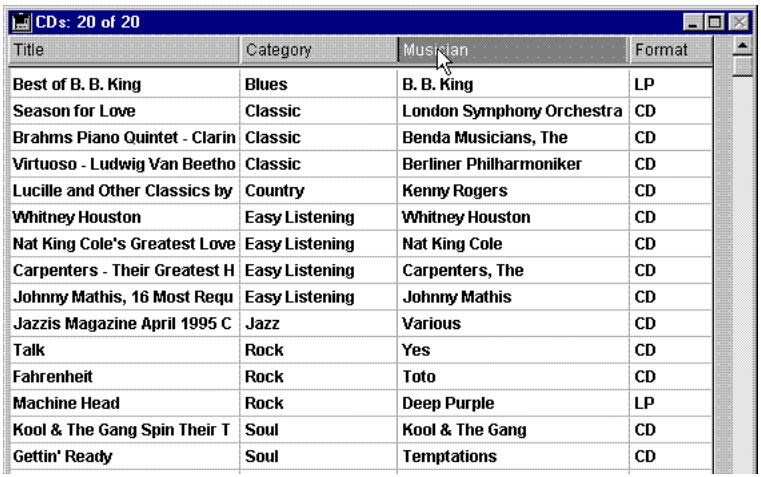
12. The following line performs a sequential sort using a related field:

```
   ` Invoices are sorted alphabetically on the Company name field
ORDER BY([Invoices];[Companies]Name;>)
```

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.

Example

13. 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
```

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 completed, OK is set to 1. Otherwise, if the sort is interrupted, OK is set to 0 (zero).

See Also

ORDER BY FORMULA.

ORDER BY FORMULA

version 3

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 on which to set the order for each level (can be of type Alphanumeric, Real, Integer, Long Integer, Date, Time or Boolean)
> or <		→	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 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 completed, OK is set to 1. Otherwise, if the sort is interrupted, OK is set to 0 (zero).

4D Server: Since expression cannot be interpreted by 4D Server, each record is sent to the local workstation; the order formula is evaluated on the workstation. This will make the order inefficient. Use the ORDER BY command whenever possible.

Unlike ORDER BY, ORDER BY FORMULA always performs a sequential sort.

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);<)
```

See Also

ORDER BY.

QUERY ({aTable}{; queryArgument{; *}})

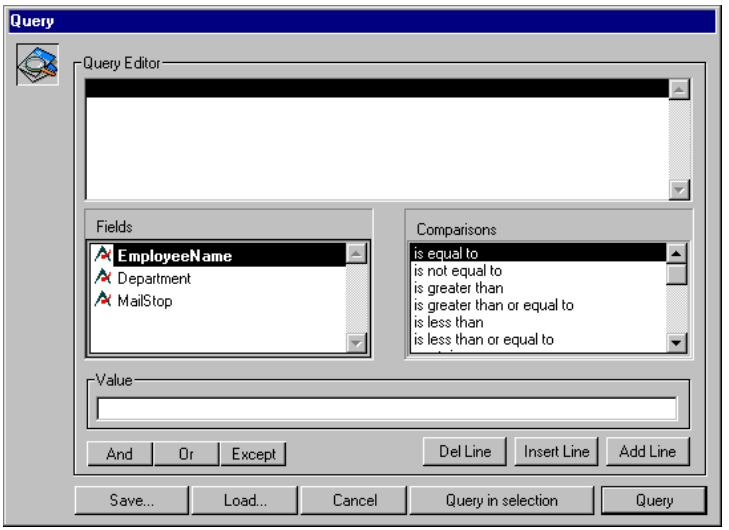
Parameter	Type		Description
aTable	Table	→	Table for which to return a selection of records, or Default table, if omitted
queryArgument		→	Query argument
*		→	Continue query flag

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).

Examples

1. The following line displays the Query editor for the [Products] table:

```
QUERY([Products])
```

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.

Examples

3. The following line looks for the [People] whose name starts with an “a”:

```
QUERY([People];[People]Last name="a@")
```

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
    ` No * ends the query definition and starts the actual query operation
QUERY([People]; |;[People]Name="b@")
```

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.

• The field is the field to query. The field may belong to another table if it belongs to a One table related to table with an automatic relation. The table to which QUERY is applied must be the Many table.

• 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.
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 must begin with a conjunction.
• 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 table 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 commands MESSAGES ON and MESSAGES OFF. 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.

Examples

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 ASCII codes.

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.

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.

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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:

```
   ` Find employees who make between…
QUERY ([Employee]; [Employee]Salary >= 10000; *)
QUERY ([Employee]; & ; [Employee]Salary < 50000)  ` ...$10,000 and $50,000
```

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. Although the [Dept]Name field is indexed, this is not an indexed query because the relation must be activated sequentially for each record in the [Employee] table:

```
   ` Find employees with salaries over $20,000 and...
QUERY ([Employee]; [Employee]Salary > 20000; *)
   ` ...who are in the marketing department
QUERY ([Employee]; &;[Dept]Name = "marketing")
```

17. 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.

18. 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
```

19. 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
```

See Also

Operators, QUERY SELECTION.

QUERY BY EXAMPLE ({aTable}{; }{*})

Parameter	Type		Description
aTable	Table	→	Table for which to return a selection of records, or Default table, if omitted
*		→	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:

```
INPUT FORM ([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
```

See Also

ORDER BY, QUERY.

System Variables or 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 ({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 table. QUERY 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 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.

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.

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.

Examples

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:

```
   ` Find the invoices entered in December
QUERY BY FORMULA ([Invoice]; Month of ([Invoice]Entered) = 12)
```

2. This example finds records for all the people who have names with more than ten characters:

```
   ` Find names longer than ten characters
QUERY BY FORMULA ([People]; Length ([People]Name)>10)
```

See Also

QUERY, QUERY BY SQL, QUERY SELECTION, QUERY SELECTION BY FORMULA.

QUERY SELECTION ({aTable}{; queryArgument{; *}})

Parameter	Type		Description
aTable	Table	→	Table for which to return a selection of records, or Default table, if omitted
queryArgument		→	Query argument
*		→	Continue query flag

Description

QUERY SELECTION looks for records in aTable. 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 command QUERY.

Example

This example illustrates the difference between QUERY and QUERY SELECTION. Here are two queries:

```
      ` Find ALL companies located in New York City
QUERY ([Company]; [Company]City="New York City")
      ` Find ALL companies doing Stock Exchange business
      ` no matter where they are located
QUERY ([Company]; [Company]Type Business="Stock Exchange")
```

Note that the second QUERY simply “ignores” the result of the first one. Compare this with:

```
      ` Find ALL companies located in New York City
QUERY ([Company]; [Company]City="New York City")
      ` Find companies doing Stock Exchange business
      ` and that are located in New York City
QUERY SELECTION ([Company]; [Company]Type Business="Stock Exchange")
```

QUERY SELECTION looks only among the selected records, therefore, in this example, among the companies located in New York City.

See Also

QUERY.

version 11 (Modified)

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 command QUERY BY FORMULA.

See Also

QUERY, QUERY BY FORMULA, QUERY SELECTION.

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, Subfield, 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 (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. Call SET QUERY AND LOCK(False) in order to disable this mechanism afterwards.

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 the DELETE SELECTION:

```
START TRANSACTION
SET QUERY AND LOCK(True)
QUERY([Customers];[Customers]Catégorie=“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 Handling

If the command is not called in the context of a transaction, an error is generated.

See Also

QUERY.

SET QUERY DESTINATION (destinationType{; destinationObject})

Parameter	Type		Description
destinationType	Number	→	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

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	Long Integer	0
Into set	Long Integer	1
Into named selection	Long Integer	2
Into variable	Long Integer	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 named of a named selection (existing or to be created)
3 (variable)	You pass a numeric variable (existing or to be created)

• 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.

Note: In client/server, you cannot use local/client sets (name preceeded by $ symbol) as a query destination. This type of set is created on client machines when queries are executed on the server. For more information on these types of sets, refer to the Sets section.

• 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.

Note: If the named selection does not exist beforehand, it will be created automatically at the end of the query.

• With:

SET QUERY DESTINATION(Into variable;$vlResult)

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 SELECTION BY FORMULA
- QUERY WITH ARRAY

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.

Examples

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;"Rolodex"+asRolodex
                                                                  {asRolodex})
            ` Perform the query
         QUERY([Phone Book];[Phone Book]Last name=asRolodex{asRolodex}+"@")
            ` Restore normal query mode
         SET QUERY DESTINATION(Into current selection)
            ` Next time we choose that letter, we won't perform the query again
         abQueryDone{asRolodex}:=True
      End if
         ` Use the named selection for displaying the records corresponding
         ` to the chosen letter
      USE NAMED SELECTION("Rolodex"+asRolodex{asRolodex})

   : (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
```

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;$2)
C_POINTER(${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)
                        ` Found an existing record with the same values as the
                        ` unsaved new record
                     $0:=False
                  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
` ...
```

See Also

QUERY, QUERY BY EXAMPLE, QUERY BY FORMULA, QUERY SELECTION, QUERY SELECTION BY FORMULA, QUERY WITH ARRAY, SET QUERY LIMIT.

SET QUERY LIMIT

version 6.0

SET QUERY LIMIT (limit)

Parameter	Type		Description
limit	Number	→	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 SELECTION BY FORMULA

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.

Examples

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)
```

2. See the second example for the command SET QUERY DESTINATION.

See Also

QUERY, QUERY BY EXAMPLE, QUERY BY FORMULA, QUERY SELECTION, QUERY SELECTION BY FORMULA, QUERY WITH ARRAY, SET QUERY DESTINATION.

40

Quick 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.

Examples

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)
```

2. The following statement retrieves the Quick Report stored in Field4 and displays it in MyArea:

```
QR BLOB TO REPORT(MyArea;[Table 1]Field4)
```

See Also

QR REPORT TO BLOB.

QR Count columns

QR Count columns (area) → Longint

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)
```

See Also

QR DELETE COLUMN, QR INSERT 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
```

See Also

QR INSERT COLUMN.

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.

See Also

QR New offscreen area.

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.

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..

See Also

QR Get command status, QR ON COMMAND.

QR Find column (area; expression) → Longint

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) → Longint

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 contextual menu.

You can use the constants from the QR Area Properties theme to designate the interface item:

Constant	Description
qr view menubar (1)	Display status of the menu bar (Displayed=1, Hidden=0)
qr view standard toolbar (2)	Display status of the Standard toolbar (Displayed=1, Hidden=0)
qr view style toolbar (3)	Display status of the Style toolbar (Displayed=1, Hidden=0)
qr view operators toolbar (4)	Display status of the operators toolbar (Displayed=1, Hidden=0)
qr view color toolbar (5)	Display status of the Color toolbar (Displayed=1, Hidden=0)
qr view column toolbar (6)	Display status of the Column toolbar (Displayed=1, Hidden=0)
qr view contextual menus (7)	Display status of the Contextual menu (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.

See Also

QR SET AREA PROPERTY.

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.
- if row equals -1, the title of the report is affected
- if row equals -2, the detail of the report is affected
- if row equals -3, the grand total of the report is affected
- if row is a positive integer, it designates the Subtotal (break) level that is affected.

You can use constants from the QR Rows for Properties theme to designate the row item (qr title= -1, qr detail=-2, qr grand total=-3).

border is the value that indicates which cell border is affected:
- 1 indicates the left border
- 2 indicates the top border
- 4 indicates the right border
- 8 indicates the bottom border
- 16 indicates the inside vertical border
- 32 indicates the inside horizontal border.

You can use constants from the QR Borders theme to designate the border item.

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.

See Also

QR SET BORDERS.

QR Get command status

QR Get command status (area; command{; value}) → Longint

Parameter	Type		Description
area	Longint	→	Reference of the area
command	Longint	→	Command number
value	Text | Longint	←	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.

See Also

QR EXECUTE COMMAND, QR ON COMMAND.

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 Destination theme.

The following table describes the values that can be retrieved in both type and specifics parameters:

Destination	Constant (value)	Specifics
Printer	qr printer (1)	N.A.
Text file	qr text file (2)	File pathname
4D View	qr 4D View area (3)	N.A.
4D Chart	qr 4D Chart area (4)	N.A.
HTML file	qr HTML file (5)	Pathname to the HTML file

If you pass an invalid area number, the error -9850 will be generated.

See Also

QR SET DESTINATION.

QR Get document property (area; property) → Longint

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 allows you to retrieve 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	Value
qr printing dialog	1
qr unit	2

• If property equals 1, the command applies to the display of the print dialog box.
- If value equals 1, the print dialog box is displayed prior to printing.
- If value equals 0, the print dialog box is not displayed prior to printing.
The default value is 1.

• If property equals 2, the command applies to the document unit.
- If value equals 0, the document unit is points.
- If value equals 1, the document unit is centimeters.
- If value equals 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.

See Also

QR SET DOCUMENT PROPERTY.

QR Get drop column

QR Get drop column (area) → Longint

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.

See Also

QR DELETE COLUMN.

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	Real	←	Header or footer height
picture	Picture	←	Picture to display
pictAlignment	Longint	←	Alignment attribute for the picture

Description

The QR GET HEADER AND FOOTER command allows you to retrieve 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 0, the picture is aligned to the left.
• If pictAlignment returns 1, the picture is centered.
• If pictAlignment returns 2, 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))
```

See Also

QR SET HEADER AND FOOTER.

QR Get HTML template (area) → Text

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.

See Also

QR SET HTML TEMPLATE.

QR GET 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	Text	←	Display format for the data

Description

List mode

The QR GET INFO COLUMN command allows you to retrieve 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 name of the actual object of the column (variable, field name or formula).

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.

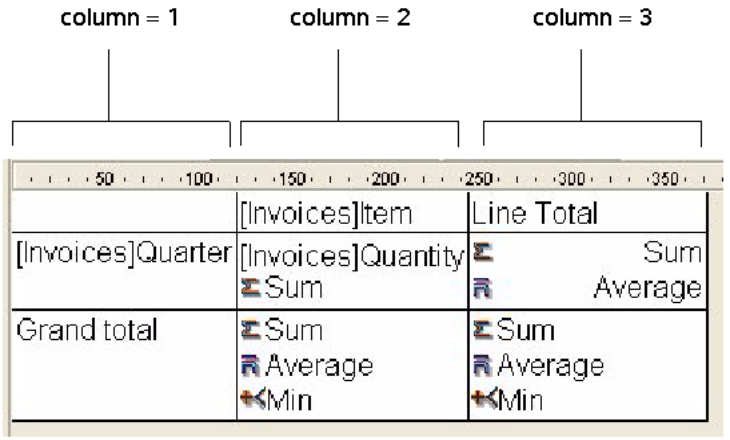
repeatedValue returns the status for data repetition. For example, if the value for a field or variable does not change from one record to the other, it may or may not be repeated when they do not change:
• if repeatedValue equals 0, values are not repeated,
• if repeatedValue equals 1, values are repeated.

format returns the display format. Display formats are the 4D formats compatible with the data displayed.

Cross-table mode

The QR GET INFO COLUMN command allows you to retrieve 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:

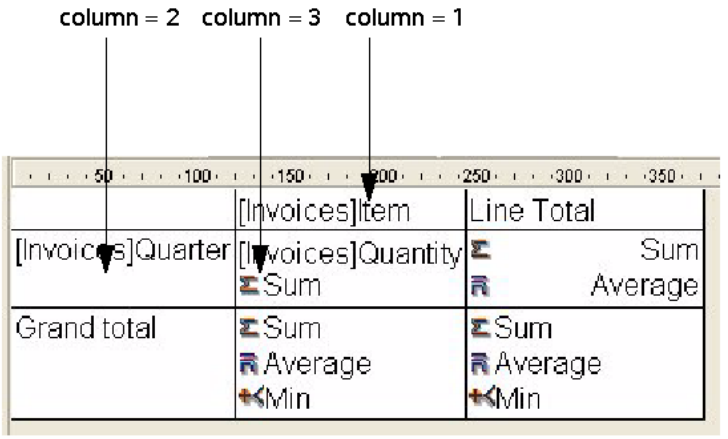


The following statement sets the size to automatic for all the columns in a cross-table report and leaves other elements unchanged:

```
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.

See Also

QR Get info row, QR SET INFO COLUMN, QR SET INFO ROW.

QR Get info row (area; row) → Longint

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:
• if row equals -1, the title display attribute is retrieved
• if row equals -2, the detail display attribute is retrieved
• if row equals -3, the grand total display attribute is retrieved
• if row is a positive integer, it designates the subtotal (break level) whose display attribute is retrieved.

You can use constants from the QR Rows for Properties theme to designate the row item (qr title= -1, qr detail=-2, qr grand total=-3)

The function result specifies whether the row is displayed or hidden. If it equals 1, the row is set to hidden; if it equals 0, the row is set to 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.

See Also

QR GET INFO COLUMN, QR SET INFO COLUMN, QR SET INFO ROW.

QR Get report kind

QR Get report kind (area) → Longint

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	Value
qr list report	1
qr cross report	2

If you pass an invalid area number, the error -9850 will be generated.

See Also

QR SET REPORT KIND.

QR Get report table (area) → Longint

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.

See Also

QR SET REPORT TABLE.

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.

See Also

QR SET SELECTION.

QR GET SORTS (area; aColumns{; aOrders})

Parameter	Type		Description
area	Longint	→	Reference of the area
aColumns	Array real	←	Sorted columns
aOrders	Array real	←	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.

See Also

QR SET SORTS.

QR Get text property

QR Get text property (area; colNum; rowNum; property) → Longint

Parameter	Type		Description
area	Longint	→	Reference of the area
colNum	Longint	→	Column number
rowNum	Longint	→	Break number
property	Longint	→	Operator value for the cell
Function result	Longint	←	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.
- if rowNum equals -1, it designates the column title.
- if rowNum equals -2, it designates the detail area.
- if rowNum equals -3, it designates the column grand total.
- if rowNum equals -4, it designates the page header.
- if rowNum equals -5, it designates the page footer.

Note: When passing -4 or -5 as rowNum, you still need to pass a column number in colNum, even if it is not used.
- if rowNum is a positive value, it designates the corresponding subtotal (break level).

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 (value)	Returned value
qr font (1)	font number as returned through Font number
qr font size (2)	font size expressed in points (9 to 255)
qr bold (3)	Bold style attribute (0 or 1)
qr italic (4)	Italic style attribute (0 or 1)
qr underline (5)	font Underline style attribute (0 or 1)
qr text color (6)	font Color attribute (color number)
qr justification (7)	font Justification attribute (0 for default, 1 for left, 2 for center or 3 for right).
qr background color (8)	background color
qr alternate background color (9)	alternate background color

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.

See Also

QR SET TEXT PROPERTY.

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 allows you to retrieve 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):
- Subtotal: between 1 and the number of Subtotal/sort.
- Grand total: -3 / constant: qr grand total.

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	Value
qr sum	1
qr average	2
qr min	4
qr max	8
qr count	16
qr standard deviation	32

If the value returned equals 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 allows you to retrieve the details of a specific cell.

area is the reference of the Quick Report area.

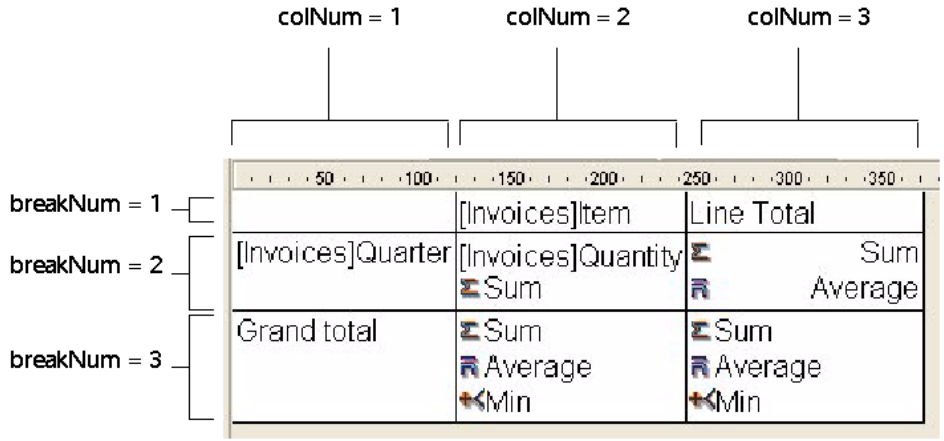
colNum is the column number of the cell whose data is going to be retrieved.

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.

See Also
QR SET TOTALS DATA.

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 at the top of the break level, <0=proportional increase

Description

The QR GET TOTALS SPACING command allows you to retrieve a space above 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 equals 0, no space is added.
• If value equals 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% above 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.

See Also

QR SET TOTALS SPACING.

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.

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)
```

See Also

QR DELETE COLUMN.

QR New offscreen area

QR New offscreen area → Longint

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

QR DELETE OFFSCREEN AREA.

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.

Note: This command does not work with external windows in Design mode.

If area equals zero, methodName will apply to each Quick Report area until the database is closed or until the following call to QR ON 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).

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.

See Also

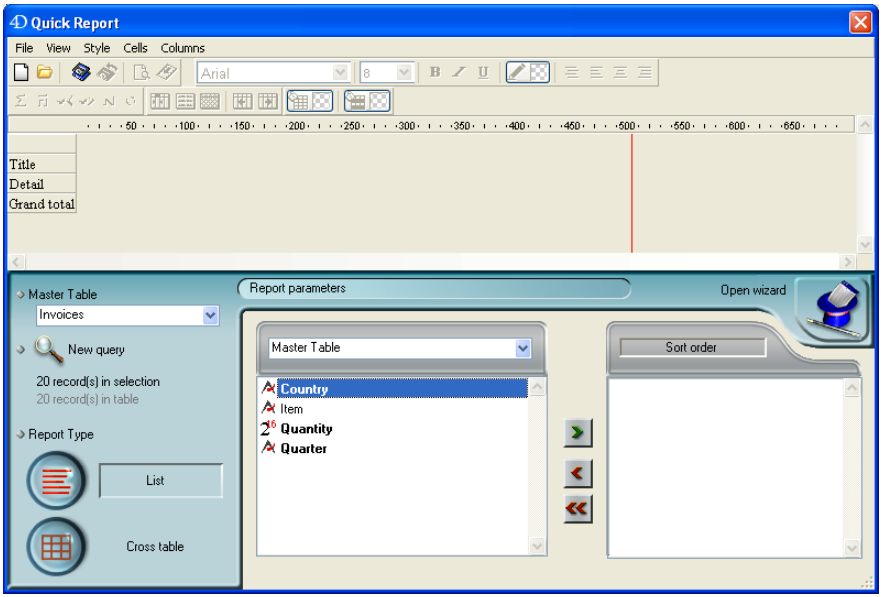
QR EXECUTE COMMAND, QR Get command status.

QR REPORT ({aTable; }document{; hierarchical{; wizard{; search}}}{; *})

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)
*	*	→	Deletion of printing dialog boxes

Description

QR REPORT prints a report for aTable, created with the Quick Report editor shown here.



The Quick Report editor allows users to create their own reports. See the *4D Design Reference* manual for details on creating reports with the Quick Report editor.

Notes:
• The editor does not appear if the table has been declared “Invisible.”
• When the editor is called using the QR REPORT command, the **All relations in automatic** option that is used to modify the automatic/manual status of the relations is hidden. This allows the developer to manage this status himself using the SET AUTOMATIC RELATIONS and SET FIELD RELATION command.

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.

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).

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 * 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.

Examples

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
```

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
```

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
```

4. Refer to the example of the SET FIELD RELATION command.

See Also

PRINT LABEL, PRINT SELECTION, SET ALLOWED METHODS.

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)
```

See Also

QR BLOB TO REPORT.

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.

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 allows you to display or hide 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 contextual menu.
You can use the constants from the QR Area Properties theme to designate the interface item:

Constant	Description
qr view menubar (1)	Display status of the menu bar (Displayed=1, Hidden=0)
qr view standard toolbar (2)	Display status of the Standard toolbar (Displayed=1, Hidden=0)
qr view style toolbar (3)	Display status of the Style toolbar (Displayed=1, Hidden=0)
qr view operators toolbar (4)	Display status of the Operators toolbar (Displayed=1, Hidden=0)
qr view color toolbar (5)	Display status of the Color toolbar (Displayed=1, Hidden=0)
qr view column toolbar (6)	Display status of the Column toolbar (Displayed=1, Hidden=0)
qr view contextual menus (7)	Display status of the Contextual menu (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.

See Also

QR Get area property.

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 allows you to set 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.
- if row equals -1, the title of the report is affected
- if row equals -2, the detail of the report is affected
- if row equals -3, the grand total of the report is affected
- if row is a positive integer, it designates the Subtotal (break) level that is affected.

You can use constants from the QR Rows for Properties theme to designate the row item. (qr title= -1, qr detail=-2, qr grand total=-3).

border is a composite value that indicates which borders of the cell are to be affected:
- 1 indicates the left border
- 2 indicates the top border
- 4 indicates the right border
- 8 indicates the bottom border
- 16 indicates the inside vertical border
- 32 indicates the inside horizontal border.

You can use constants from the QR Borders theme to designate the border item.

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.

See Also

QR GET BORDERS.

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.

The following table describes the values that can be passed in both type and specifics parameters:

Destination	Constant (value)	specifics
Printer	qr printer (1)	N.A.
Text file	qr text file (2)	Pathname to the file
4D View	qr 4D View area (3)	N.A.
4D Chart	qr 4D Chart area (4)	N.A.
HTML file	qr HTML file (5)	Pathname to the HTML file

Text file (2): If you pass an empty string as the file's pathname, 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.

4D View (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.

4D Chart(4): A 4D Chart external window is created and populated with the results of the current settings of the Quick Report area. For detailed information on how the translation is performed, please refer to the the *Design Reference* manual.

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 you pass an invalid destination value, 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; 2; "MyDoc.txt")
QR RUN(MyArea)
```

See Also

QR GET DESTINATION.

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 allows you to display the printing dialog or to define the unit used for the document.

In property, you can pass the following constants, located in the QR Document properties constant theme:

Constant	Value
qr printing dialog	1
qr unit	2

• If property equals 1, the command applies to the display of the print dialog.
- If value equals 1, the print dialog is displayed prior to printing.
- If value equals 0, the print dialog is not displayed prior to printing (default value).

• If property equals 2, the command applies to the document unit.
- If value equals 0, the document unit is points.
- If value equals 1, the document unit is centimeters.
- If value equals 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.

See Also

QR Get document property.

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	Real	→	Header or footer height
picture	Picture	→	Picture to display
pictAlignment	Longint	→	Alignment attribute for the picture

Description

The QR SET HEADER AND FOOTER command allows you to set 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 will be affected;
• if selector equals 2, the footer will be 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 equals 0, the picture is aligned to the left.
• If pictAlignment equals 1, the picture is centered.
• If pictAlignment equals 2, 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)
```

See Also

QR GET HEADER AND FOOTER.

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 ().

<!--#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.

See Also

QR Get HTML template.

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 allows you to set 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 displayed or hidden:
• if hide equals 1, the column is set to hidden;
• if hide equals 0, the column is set to displayed.

size is the size in pixels to assign to the column. If size equals -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;"###,##")
```

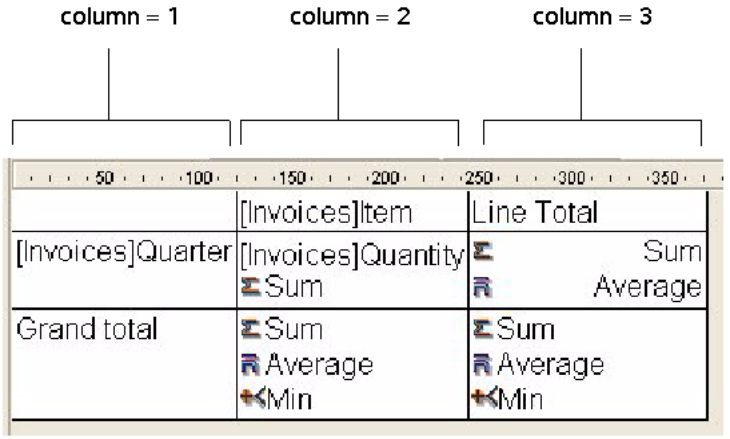
Cross-table mode

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.

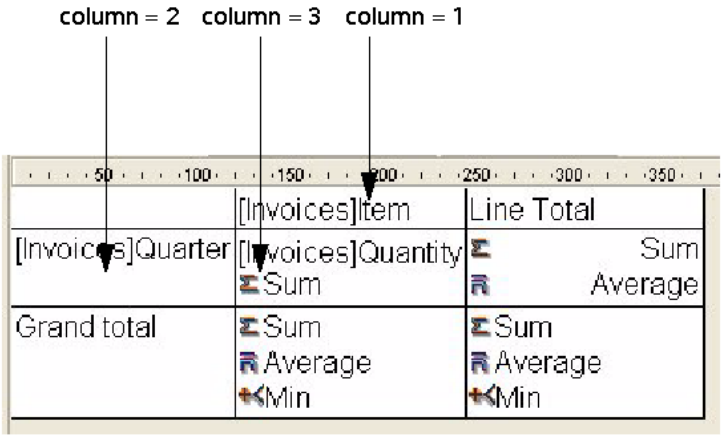


The following statement will set the size to automatic for all the columns in a cross-table report and will leave other elements unchanged:

```
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:

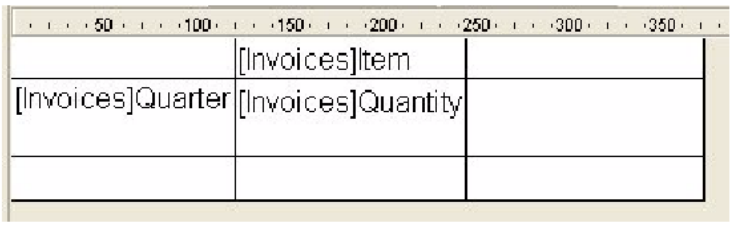


You will notice that not all cells can be addressed using the QR SET INFO COLUMN command, the cells that are not numbered above are addressed using QR SET TOTALS DATA.

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.

See Also

QR GET INFO COLUMN, QR Get info row, 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 displays/hides the row whose reference was passed in row.

row designates which row is affected:
- if row equals -1, the title of the report is affected,
- if row equals -2, the detail of the report is affected,
- if row equals -3, the grand total of the report is affected,
- if row is a positive integer, it designates the subtotal (break) level that is affected.

You can use constants from the QR Rows for Properties theme to designate the row item (qr title= -1, qr detail=-2, qr grand total=-3).

hide specifies whether the line is displayed or hidden:
- if hide equals 1, the row is set to hidden;
- if hide equals 0, the row is set to 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.

Example

The following statement hides the detail row:

```
QR SET INFO ROW (area;qr detail; 1)
```

See Also

QR GET INFO COLUMN, QR Get info row, QR SET INFO COLUMN.

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	Value
qr list report	1
qr cross report	2

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.

See Also

QR Get report kind.

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.

See Also

QR Get report table.

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 allows you to highlight a cell, a row, a column or the entire area as you would with a mouse click. It also allows you to deselect the current selection.

left is the number of the left boundary. If left equals 0, the entire row is selected.
top is the number of the top boundary. If top equals 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 equal 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.

See Also

QR GET SELECTION.

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 allows you to set 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{$i} equals 1, the sort order is ascending.
• If aOrders{$i} equals - 1, the sort order is descending.

Cross-table mode

In the case of cross-table mode, 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.

See Also

QR GET SORTS.

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	→	Operator value for the cell
value	Longint	→	Value for the selected property

Description

The QR SET TEXT PROPERTY command allows you to set 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:
• if rowNum equals -1, it designates the column title.
• if rowNum equals -2, it designates the detail area.
• if rowNum equals -3, it designates the column grand total.
• if rowNum equals -4, it designates the page header.
• if rowNum equals -5, it designates the page footer.
You can use constants from the QR Rows for Properties theme to designate the row item. Constant Values are: qr title (-1), qr detail (-2), qr grand total (–3), qr header (–4), and qr footer (-5).
Note: When passing -4 or -5 as rowNum, you still need to pass a column number in colNum, even if it is not used.
• if rowNum is a positive value, it designates the corresponding subtotal (break level).

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 (value)	Value to set
qr font (1)	font number as returned through FONT LIST
qr font size (2)	font size expressed in points (9 to 255)
qr bold (3)	Bold style attribute (0 or 1)
qr italic (4)	Italic style attribute (0 or 1)
qr underline (5)	font Underline style attribute (0 or 1)
qr text color (6)	font Color number attribute (longint)
qr justification (7)	font Justification attribute (0 for default, 1 for left, 2 for center or 3 for right)
qr background color (8)	background color number
qr alternate background color (9)	alternate background color number

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:

```
   `The following call assigns the font Times:
QR SET TEXT PROPERTY(qr_area;1;-1;qr font;Font number("Times"))
   `assigning the font size 10 points:
QR SET TEXT PROPERTY(qr_area;1;-1;qr font size;10)
   `assigning the font attribute Bold:
QR SET TEXT PROPERTY(qr_area;1;-1;qr bold;1)
   `assigning the font attribute Italic:
QR SET TEXT PROPERTY(qr_area;1;-1;qr italic;1)
   `assigning the font attribute Underline:
QR SET TEXT PROPERTY(qr_area;1;-1;qr underline;1)
   `assigning the color bright green:
QR SET TEXT PROPERTY(qr_area;1;-1;qr text color;0x0000FF00)
```

See Also

QR Get text property.

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 allows you to set 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.

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	Value
qr sum	1
qr average	2
qr min	4
qr max	8
qr count	16
qr standard deviation	32

If operator equals 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 allows you to set the details of a specific cell.

area is the reference of the Quick Report area.

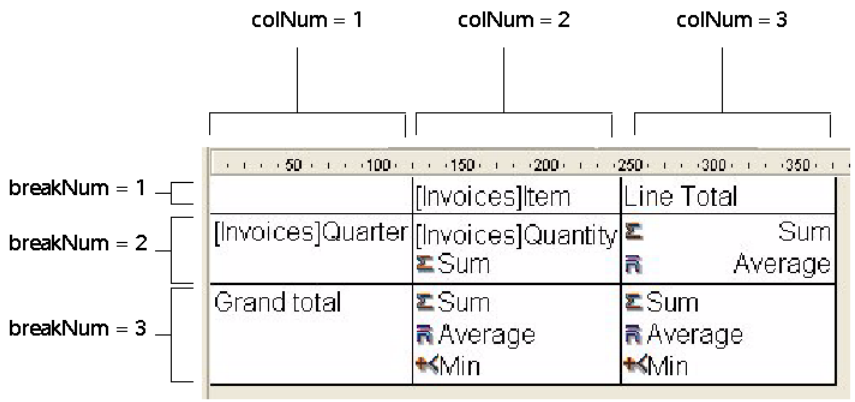
colNum is the column number of the cell whose data is going to be set.

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.

See Also

QR GET TOTALS DATA.

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 at the top of the break level, <0=proportional increase

Description

The QR SET TOTALS SPACING command allows you to set a space above 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 equals 0, no space is added.
• If value equals 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% above the subtotal row.

Note: If the space above a subtotal row "pushes" the row to the next 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.

See Also

QR GET TOTALS SPACING.

41

Record Locking

4D Developer and 4D Server/4D Client 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:
• Each table is in either a read-only or a read/write state.
• Records become locked when they are loaded and unlocked when they are unloaded.
• 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. 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, the record is always locked. In other words, the record can be displayed, printed, and otherwise used, but it cannot be modified.

Note that read-only status applies only to editing existing records. Read-only status does not affect the creation of new records. You can add records to a read-only table using CREATE RECORD and ADD RECORD or the menu commands of the Design environment.

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
- 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, the 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 or 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 the 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.

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 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, then a record loaded from that table is locked. A locked record cannot be saved or deleted. Read-only is the preferred state, because it allows other users to load, modify, and then save the record.

If a table is in the read/write state, then a record that is loaded from that table is unlocked only if no other users have locked the record first. An unlocked record can be modified and saved. A table should be put into the read/write state before a record needs to be loaded, modified, and then saved.

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 ATTRIBUTES command to see which user and/or process have locked a record.

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 ATTRIBUTES([Commands];$Process;$User;$Machine;$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.

See Also

LOAD RECORD, Locked, LOCKED ATTRIBUTES, Methods, READ ONLY, Read only state, READ WRITE, UNLOAD RECORD, Variables.

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.

See Also

Locked, Record Locking, UNLOAD RECORD.

Locked {(aTable)} → Boolean

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 is locked by another user or process and cannot be saved. 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 ATTRIBUTES command. If the record has been deleted, the LOCKED ATTRIBUTES command returns -1 in the process parameter.

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.

See Also

LOAD RECORD, LOCKED ATTRIBUTES, Record Locking.

LOCKED ATTRIBUTES ({table; }process; 4Duser; sessionUser; processName)

Parameter	Type		Description
aTable	Table	→	Table to check for record locked, or Default table, if omitted
process	Number	←	Process reference number
4Duser	String	←	4D user name
sessionUser	String	←	Name of user that opened work-session
processName	String	←	Process name

Description

LOCKED ATTRIBUTES returns information about the user and process that have locked a record. The process number (on the server machine), 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.

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.

In single-user mode, this command returns values in process and processName only if a record is locked. The values returned in 4Duser and sessionUser are empty strings.

In Client/Server mode, the returned process number is the number of the process on the server.

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 in the 4D Server administration window for each open process).

See Also

Locked, Record Locking.

READ ONLY {(aTable | *)}

Parameter	Type		Description
aTable | *	Table	→	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.

See Also

Read only state, READ WRITE, Record Locking.

READ WRITE {(aTable | *)}

Parameter	Type		Description
aTable | *	Table	→	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.

See Also

READ ONLY, Read only state, Record Locking.

UNLOAD RECORD

UNLOAD RECORD {(table)}

Parameter	Type		Description
table	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).

See Also

LOAD RECORD, Record Locking.

42

Records

About Record Numbers

version 11 (Modified)

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.

Note: Temporary record numbers that are created during transactions start at 18,000,000.

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 nonrepeating number that may be assigned to a field of a record. It is not automatically stored with each record. It starts 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.

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

See Also

Record number, Selected record number, Sequence number.

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.

Note to version 3 users: While entering a record, if you have to check a multiple field unique value, use the new 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.

See Also

POP RECORD, PUSH RECORD, SET QUERY DESTINATION.

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 table, 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.

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
```

See Also

ADD RECORD, MODIFY RECORD, SAVE 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.

Note: 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.

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
```

See Also

Locked, Triggers.

DISPLAY RECORD

version 3

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
INPUT FORM ([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
```

See Also

MESSAGE.

DUPLICATE RECORD

version 3

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.

See Also

SAVE RECORD.

GOTO RECORD

version 3

GOTO RECORD ({aTable; }record)

Parameter	Type		Description
aTable	Table	→	Table in which to go to the record, or Default table, if omitted
record	Number	→	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, the selection becomes empty.

Example

See the example for Record number.

See Also

About Record Numbers, Record number.

Is new record {(aTable)} → Boolean

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 the 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 Developer or 4D Client. In single-user version, the command returns False (the record is considered as already created). In client/server version, 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 instructions 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
```

See Also

Modified record, Record number.

Is record loaded

version 6.5

Is record loaded {(aTable)} → Boolean

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 the aTable's current record is loaded in the current process.

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

version 3

Modified record {(aTable)} → Boolean

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.

Example

The following example shows a typical use for Modified record:

```
If (Modified record ([Customers]))
   SAVE RECORD ([Customers])
End if
```

See Also

Modified, Old, SAVE RECORD.

POP RECORD

version 3

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
```

See Also

PUSH RECORD, Using the Record Stack.

PUSH RECORD

version 3

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.

Example

The following example pushes the record for the customer onto the record stack:

```
PUSH RECORD ([Customer]) ` Push customer's record onto stack
```

See Also

POP RECORD, Using the Record Stack.

Record number

version 3

Record number {(table)} → Number

Parameter	Type		Description
aTable	Table	→	Table for which to return the number of the current record, or Default table, if omitted
Function result	Number	←	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. Record numbers will also change if you compact the database.

4D Server: This command returns a different result for the On Validate form event depending on whether it is executed on 4D Developer or 4D Client. In single-user version, the command returns a record number (the record is considered as already created). In client/server version, 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
```

See Also

About Record Numbers, GOTO RECORD, Is new record, Selected record number, Sequence number.

Records in table

version 3

Records in table {(table)} → Number

Parameter	Type		Description
aTable	Table	→	Table for which to return the number of records, or Default table, if omitted
Function result	Number	←	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.")
```

See Also

Records in selection.

SAVE RECORD {(table)}

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.

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 ADD SUBRECORD, CREATE SUBRECORD, or 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
```

See Also

CREATE RECORD, Locked, Triggers.

Sequence number

version 2004.1 (Modified)

Sequence number {(aTable)} → Number

Parameter	Type		Description
aTable	Table	→	Table for which to return the sequence number, or Default table, if omitted
Function result	Number	←	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. By default, the numbering starts at 1. You can change the numbering for a table using the SET DATABASE PARAMETER command.

You should use the Sequence number function instead of the #N symbol if:
- You are creating records procedurally
- The sequence number needs to start at a number other than 1
- 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 is the same number assigned by using the #N symbol as the default value for a field in a form. For information on assigning default values, see the *4D Design Reference* manual.

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 record, 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
```

See Also

About Record Numbers, Record number, Selected record number.

43

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

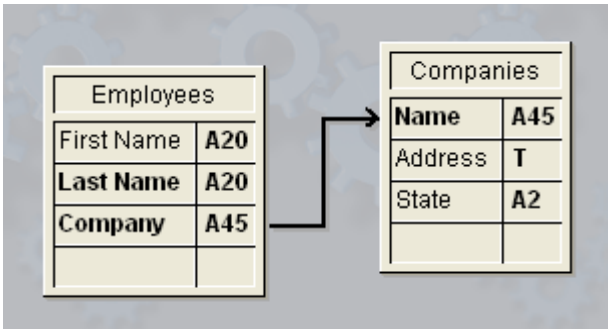
Two tables can be related with automatic table relations. In general, when an automatic table relation is established, it loads or selects the related records in a related table. Many operations cause the relation to be established.

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
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
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.

See Also

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.

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.

See Also

SAVE RELATED ONE.

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.

See Also

GET FIELD RELATION, GET RELATION PROPERTIES, SET AUTOMATIC RELATIONS.

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
*	*	→	• 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.

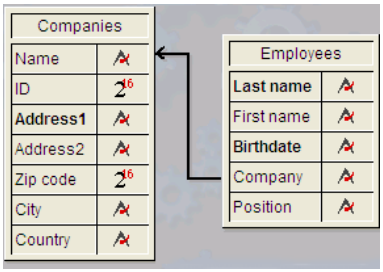
You can compare the values returned in the one and many parameters with the constants of the “Relations” theme:

Constant	Type	Value
No relation	Longint	0
Structure configuration	Longint	1
Manual	Longint	2
Automatic	Longint	3

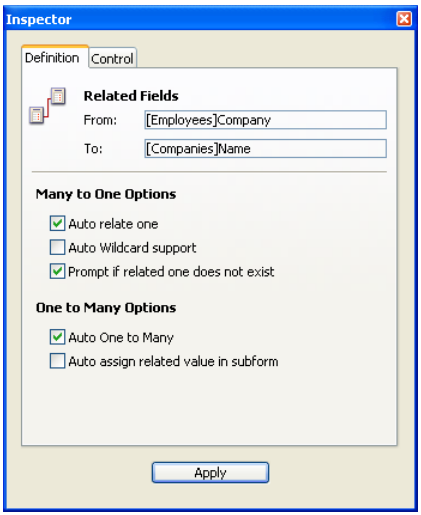
• The optional * parameter lets you “force” the reading of the current status of the relation, even if it has not been modified by programming. In other words, when you pass the * parameter, only the values 2 or 3 can be returned in the one and many parameters.

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, SET FIELD RELATION and 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)

GET FIELD RELATION([Employees]Company;one;many) `returns 2,1
GET FIELD RELATION([Employees]Company;one;many;*) `returns 2, 2

   `re-establishes the parameters set in Design environment for Many-to-One relation
SET FIELD RELATION ([Employees]Company;1;0)

GET FIELD RELATION([Employees]Company;one;many)  `returns 1,1
GET FIELD RELATION([Employees]Company;one;many;*)  `returns 3,2

   `changes all relations of all tables to automatic
SET AUTOMATIC RELATIONS(True;True)

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
```

See Also

GET AUTOMATIC RELATIONS, GET RELATION PROPERTIES, SET AUTOMATIC RELATIONS, SET FIELD RELATION.

OLD RELATED MANY

OLD RELATED MANY (field)

Parameter	Type		Description
field	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.

See Also

OLD RELATED ONE, RELATE MANY.

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.

See Also

Old, OLD RELATED MANY, RELATE ONE, SAVE RELATED ONE.

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.

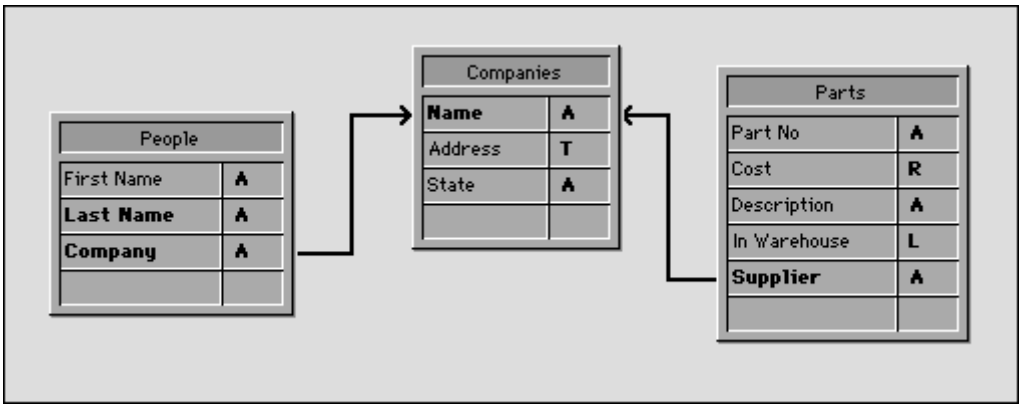
The first form, RELATE MANY(oneTable), establishes all One-to-Many relations for oneTable. It changes the current selection for each table that has a One-to-Many relation to oneTable. The current selections in the Many tables depend on the current value of each related field in the One table. Each time this command is executed, the current selections of the Many tables will be regenerated.

The second form, RELATE MANY(oneField), establishes the One-to-Many relation for oneField. It changes the current selection for only those tables that have relations with oneField. This means that the related records become the current selection for the Many table.

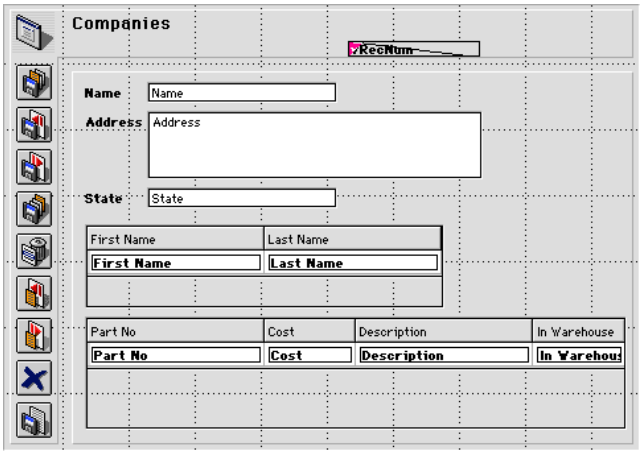
Note: If the current selection in the One table is empty while the RELATE MANY command is executed, it has no effect.

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 ALERTcommand 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
```

See Also

OLD RELATED MANY, RELATE ONE.

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.

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)
```

See Also

QUERY, RELATE ONE, RELATE ONE SELECTION.

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 can be specified only if manyField is an Alpha field. The choiceField must be a field in the related table. The choiceField must be an Alpha, Numeric, Date, Time, or Boolean field; it cannot be a text, picture, BLOB, or subtable field.

If choiceField is specified and more than one record is found in the related table, RELATE ONE displays a selection list of records that match the value in manyField. In the 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. 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.

RELATE ONE works with relations to subtables, but you must have a relation to the parent table and to the subtable's related field in order for the relation to be properly established. When using a relation to a subrecord, you must first use RELATE ONE to load the related record into memory, then use a second RELATE ONE command for the subtable.

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]ID, and the other relation is from [Invoice]Ship to to [Customers]ID.

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
```

See Also

OLD RELATED ONE, RELATE MANY.

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.

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.

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])
```

See Also

QUERY, RELATE MANY SELECTION, RELATE ONE, Sets.

SAVE RELATED ONE (aField)

Parameter	Type		Description
aField	Field	→	Many field

Description

SAVE RELATED ONE saves the record related to aField. Execute a SAVE RELATED ONE command to update a record created with CREATE RELATED ONE, or to save modifications to a record loaded with RELATE ONE.

SAVE RELATED ONE does not apply to subtables, because saving the parent record automatically saves the subrecords.

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.

See Also

CREATE RELATED ONE, Locked, RELATE ONE, Triggers.

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. 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.

Relations that are set as automatic in the Design environment are not affected by this command.

If all relations have been set as manual in the Design environment, this command makes it possible to change them to be automatic, just before executing operations that need the relation to be automatic (such as relational searches and sorts). After the operation is finished, the relation can be changed back to manual.

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)
```

See Also

GET AUTOMATIC RELATIONS, GET RELATION PROPERTIES, Relations, SELECTION RANGE TO ARRAY, SELECTION TO ARRAY, SET FIELD RELATION.

SET FIELD RELATION (manyTable | manyField; one; many)

Parameter	Type		Description
manyTable | manyField	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 lets you set the automatic/manual status of each relation of the database separately for the current process, regardless of its initial status as defined 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 will only apply to the relation starting from the specified Many field.
• If you pass a table name (manyTable), the command will apply to all the relations starting from the specified Many table.
• If there is no relation starting from the manyField field or manyTable table, the one and many parameters return 0, 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.

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)
```

See Also

GET AUTOMATIC RELATIONS, GET FIELD RELATION, GET RELATION PROPERTIES, SET AUTOMATIC RELATIONS.

44

Resources

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.

New resource management principles

In 4D v11, the notion of “resources” must now be understood in the broader sense as “files that are necessary for the translation of application interfaces.” The new architectue of resources is based on a folder, named **Resources**, that must be located next to the database structure file (.4db or .4dc). It is not created by default; you will have to create it if your database uses resources. 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). Two commands of the "Resources" theme can be used to take advantage of this architecture (see the Get indexed string and STRING LIST TO ARRAY commands).

Resource management (traditional principles)

Compatibility Note: The traditional resource management principles are progressively being abandoned in 4D (see previous paragraph). For new databases, it is now recommended to rely on XLIFF architecture or the use of standard files.

What is a resource?

A **resource** is data of any kind stored in a defined format in a separate file or in the **resource fork** of a Macintosh file. Resources typically include data such as strings, pictures, icons and so on. As a matter of fact, you can create and use your own kinds of resources and store whatever data you want into them.

Data Fork, Resource Fork and Resource file

Originally, on Macintosh, data and resources were stored in the same file, made of a **data fork** and a **resource fork**. The data fork of a Macintosh file is the equivalent of a file on Windows and UNIX. The resource fork of a Macintosh file contains the Macintosh-based resources of the file and has no direct equivalent on Windows or UNIX.

Although this feature is still supported by 4D, now under Mac OS as well as under Windows, the resources are stored in a separate file (in the data fork on Mac OS). This principle is managed transparently by 4D and allows direct exchange of files between the different platforms without conversion.

Resource file management commands (Create resource file and Open resource file) can work directly within the data fork for a better cross-platform compatibility.

Resource Files

In databases created with a version of 4D prior to v11, 4D automatically created a .rsr file in order to store the structure file resources and a .4dr file for the data file resources.
The 4D application itself uses resources, stored in a file suffixed ".RSR". 4D Plug-ins like 4D Write can also use resources.

Creating Your Own Resource Files

In addition to the resource files provided by 4D, you can create and use your own resource files using the 4D commands Create resource file and Open resource file. These two commands return a **resource file reference number** that uniquely identifies the open resource file. The resource file reference number is the equivalent of the document reference number for regular files returned by System documents commands such as Open document. All the 4D Resources commands optionally expect a resource file reference number. After you have finished with a resource file, remember to close it using the command CLOSE RESOURCE FILE.

The Resource Files Chain

When you work with a 4D database, you can either work with **all the currently open resource files** or with a **specific resource file**.

Multiple resource files can be open at the same time. This is always the case from within a 4D database. The following files are open:

- On Macintosh, the System resource file.
- On Windows, the ASIPORT.RSR file (it contains part of the Macintosh system resources).
- The 4D application resource file.
- The database structure resource file (if any).
- The database data file resource file (if any) may be optionally open.
- Finally, you can open your own resource file using the command Open resource file.

This list of open resource files is called the **resource files chain**. You can search for a given resource in two ways:

- If you pass a resource file reference number to a resource 4D command, the resource is searched for in that resource file only.
- If you do not pass a resource file reference number to a 4D Resource command, the resource is searched for in all currently open resource files, starting with the most recently opened file and ending with the first opened file. The resource files chain is thus browsed in the reverse order of opening—the last opened resource file is examined first.

Here is an example:

```
$vhResFile:=Create resource file("Just_a_file")
If (OK=1)
    ARRAY STRING(63;asSomeStrings;0)
    STRING LIST TO ARRAY(8;asSomeStrings;$vhResFile)
    ALERT("The size of the array is "+String(Size of array(asSomeStrings))+" element(s).")
    STRING LIST TO ARRAY(8;asSomeStrings)
    ALERT("The size of the array is "+String(Size of array(asSomeStrings))+" element(s).")
    CLOSE RESOURCE FILE($vhResFile)
End if
```

At execution of this method, the first alert will display “The size of the array is 0 element(s)” and the second alert will display “The size of the array is 634 element(s)”.

The first call:

```
STRING LIST TO ARRAY(8;asSomeStrings;$vhResFile)
```

looks for the resource "STR#" ID=8 only in the resource file just created and open by the call to Create resource file. Because the file is new and therefore empty, the resource is not found.

The second call:

```
STRING LIST TO ARRAY(8;asSomeStrings)
```

looks for the resource "STR#" ID=8 in all the currently open resource files. Since the file just created and opened (by the call to Create resource file) does not contain that resource, STRING LIST TO ARRAY then looks for the resource in the database structure resource file. This resource file does not contain that resource either, so STRING LIST TO ARRAY then examines the 4D resource file, locates the resource in this file, and populates the array with it.

Conclusion: When working with resource files, if you want to access a specific file, make sure to pass the resource file reference number to a 4D Resources command. Otherwise, the command assumes that you do not care which file is the source of the resources.

Resource Type

A resource file is highly structured. In addition to the data of each resource, it contains a header and a map that fully describe its contents.

Resources are classified by **types**. A resource type is always denoted by a 4-character string. A resource type is both case sensitive and diacritical sensitive. For example, the resource types “Hi_!”, “hi_!” and “HI_!” are all different.

Important: Resource types with lowercase characters are reserved for use by the Operating System. Avoid designating your own resource types with lowercase characters.

The following is a list of some commonly-used resource types:

• A resource of type “STR#” is a resource containing a list of Pascal strings. This resource is called a **string list resource**.
• A resource of type “STR ” (note the space as fourth character) is a resource containing an individual Pascal string. This resource is called a **string resource**.
• A resource of type “TEXT” is a resource containing a text string without length. This resource is called a **text resource**.
• A resource of type “PICT” is a resource containing a Macintosh-based QuickDraw picture that you can use and display on both Macintosh and Windows with 4D. This resource is called a **picture resource**.

• A resource of type “cicn” is a resource containing a Macintosh-based color icon that you can use and display with 4D on both Macintosh and Windows. This resource is called a **color icon resource**. For example, a “cicn” resource can be associated with an item of a hierarchical list, using the command SET LIST ITEM PROPERTIES.

In addition to the standard resource types, you can create you own types. For example, you can decide to work with resources of type “MTYP” (for “My Type”).

To obtain the list of resource types currently present in all open resource files or in a particular resource file, use the command RESOURCE TYPE LIST. Then, to obtain the list of a specified type of resource present in all open resource files or in a particular resource file, use the command RESOURCE LIST. This command returns the IDs and Names (see next section) of all resources of a given type.

WARNING: Many applications rely on the resource type for working with its contents. For example, while accessing a “STR#” resource, applications expect to find a string list in the resource. Do NOT store inconsistent data in resources of standard types; this may lead to system errors in your 4D application or in other applications.

WARNING: A resource is a highly structured file—do NOT access the file with commands other than Resources commands. Note that nothing prevents you from passing a resource file reference number (formally a 4D time expression like the document reference number) to a command such as SEND PACKET. However, if you do so, you will probably damage the resource file.

WARNING: A resource file can contain about up to 2,700 individual resources. Do NOT attempt to exceed this limit. Note that nothing prevents you from doing so; however, this will damage the resource file and make it unusable.

Resource Name and Resource ID

A resource has a **resource name**. A resource name can be up to 255 characters, and is diacritical sensitive but not case sensitive. Resource names are useful for describing a resource, but you access a resource using its type and ID number. Resource names are not unique; several resources can have the same name.

A resource has a **resource ID number** (for short, resource ID or ID). This ID is unique within a resource type and a resource file. For example:

- One resource file can contain a resource “ABCD” ID=1 and a resource “EFGH” ID=1.
- Two resource files can contain a resource with the same type and ID.

When you access a resource using a 4D command, you indicate its type and ID. If you do not specify the resource file in which you are looking for this resource, the command returns the occurrence of the resource found in the first examined resource file. Remember that resource files are examined in the reverse order in which they have been opened.

The range of a resource ID is -32,768..32,767.

Important: Use the range 15,000..32,767 for your own resources. Do NOT use negative resource IDs; these 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.

To obtain the IDs and names of a given resource type, use the command RESOURCE LIST.

To obtain the name of an individual resource, use the command Get resource name.

To change the name of and individual resource, use the command SET RESOURCE NAME.

As each 4D command optionally accepts a resource file reference number, you can easily deal with resources having the same type and ID in two different resource files.

The following example copies all the “PICT” resources from one resource file to another:

```
    ` Open an existing resource file
$vhResFileA:=Open resource file("")
If (OK=1)
        ` Create a new resource file
    $vhResFileB:=Create resource file("")
    If (OK=1)
            ` Get the ID and Name lists of all the resources of type "PICT"
            ` located in the resource file A
        RESOURCE LIST("PICT";$aiResID;$asResName;$vhResFileA)
            ` For each resource:
        For($vlElem;1;Size of array($aiResID))
            $viResID:=$aiResID{$vlElem}
                ` Load the resource from file A
            GET RESOURCE ("PICT";$viResID;vxResData;$vhResFileA)
                ` If the resource could be loaded
            If (OK=1)
                    ` Add and write the resource into file B
                SET RESOURCE ("PICT";$viResID;vxResData;$vhResFileB)
                    ` If the resource could be added and written
                If (OK=1)
                        ` Copy also the name of the resource
                    SET RESOURCE NAME("PICT";$viResID;$asResName{$vlElem};
                                                                $vhResFileB)
                        ` As well as its properties (see Resource Properties below)
                    $vlResAttr:=Get resource properties("PICT";$viResID;$vhResFileA)
                    SET RESOURCE PROPERTIES("PICT";$viResID;$vlResAttr;$vhResFileB)
                Else
                    ALERT("The resource PICT ID="+String($viResID)+
                                                        " could not be added.")
                End if
            Else
                ALERT("The resource PICT ID="+String($viResID)+" could not be loaded.")
            End if
        End for
        CLOSE RESOURCE FILE($vhResFileB)
    End if
    CLOSE RESOURCE FILE($vhResFileA)
End if
```

Resource Properties

Besides its type, name and ID, a resource has additional **properties** (also called attributes). For example, a resource may or may not be purged. This attribute tells the Operating System whether or not a loaded resource can be purged from memory when free memory is required for allocating another object. As shown in the previous example, when creating or copying a resource, it can be important to not only copy the resource, but also its name and properties. For a complete explanation of resource properties, see the description of the commands Get resource properties and SET RESOURCE PROPERTIES.

Handling Resource Contents

To load a resource of any type into memory, call GET RESOURCE, which returns the contents of the resource in a BLOB.

To add or rewrite a resource on disk, call SET RESOURCE, which sets the contents of the resource to the contents of the BLOB you pass.

To delete an existing resource, use the command DELETE RESOURCE.

To simplify handling of standard resource types, 4D provides additional built-in commands that save you from having to parse a BLOB in order to extract the resource data:

- STRING LIST TO ARRAY populates a String or Text array with the strings contained in a string list resource.
- ARRAY TO STRING LIST creates or rewrites a string list resource with the elements of a String or Text array.
- Get indexed string returns a particular string from a string list resource.
- Get string resource returns the string from a string resource.
- SET STRING RESOURCE creates or rewrites a string resource.
- Get text resource returns the text of a text resource.
- SET TEXT RESOURCE creates or rewrites a text resource.
- GET PICTURE RESOURCE returns the picture of a picture resource.
- SET PICTURE RESOURCE creates or rewrites a picture resource.
- GET ICON RESOURCE returns a color icon resource as a picture.

Note that these commands are provided to simplify manipulation of standard resource types; however, they do not prevent you from using GET RESOURCE and SET RESOURCE using BLOBs. For example, this line of code:

```
ALERT(Get text resource(20000))
```

is the shorter equivalent of:

```
GET RESOURCE("TEXT";20000;vxData)
If (OK=1)
    $vlOffset:=0
    ALERT(BLOB to text(vxData;UTF8 Text without length;$vlOffset;BLOB Size(vxData)))
End if
```

4D Commands and Resources

In addition to the Resources commands described in this chapter, there are other 4D commands that work with resources and resource files:

• On Macintosh, DOCUMENT TO BLOB and BLOB TO DOCUMENT can load and write the whole resource fork of a Macintosh file.
• Using the commands SET LIST ITEM PROPERTIES and SET LIST PROPERTIES, you can associate picture or color icon resources to the items of a list or use color icon resources as nodes of a list.
• The PLAY command plays “snd ” resources on both Macintosh and Windows.
• The SET CURSOR command changes the appearance of the mouse using “CURS” resources.

See Also

BLOB Commands, Get component resource ID, OS Resource Manager Errors.

ARRAY TO STRING LIST (strings; resID{; resFile})

Parameter	Type		Description
strings	String array	→	String or Text array (new contents for the STR# resource)
resID	Number	→	Resource ID number
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The ARRAY TO STRING LIST command creates or rewrites the string list (“STR#”) resource whose ID is passed in resID. The contents of the resource are created from the strings passed in the array strings. The array can be a String or Text array.

If the resource cannot be added, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top the resource files chain (the last resource file opened).

Note: Each string of a string list resource can contain up to 255 characters.

Tip: Limit your use of string list resources to resources no more than 32K in total size, and a maximum of a few hundred strings maximum per resource.

Example

Your database relies on a given set of fonts.

In the On Exit Database Method, you write:

```
    ` On Exit Database Method
If (<>vbFontsAreOK)
    FONT LIST($atFont)
    $vhResFile:=Open resource file("FontSet")
    If (OK=1)
        ARRAY TO STRING LIST($atFont;15000;$vhResFile)
        CLOSE RESOURCE FILE($vhResFile)
    End if
End if
```

In the On Startup Database Method, you write:

```
    ` On Startup Database Method
<>vbFontsAreOK:=False
FONT LIST($atNewFont)
If (Test path name("FontSet")#Is a document)
    $vhResFile:=Create resource file("FontSet")
Else
    $vhResFile:=Open resource file("FontSet")
End if
If (OK=1)
    STRING LIST TO ARRAY(15000;$atOldFont;$vhResFile)
    If (OK=1)
        <>vbFontsAreOK:=True
        For($vlElem;1;Size of array($atNewFont))
            If ($atNewFont{$vlElem}#$atOldFont{$vlElem})
                $vlElem:=MAXLONG
                <>vbFontsAreOK:=False
            End if
        End for
    Else
        <>vbFontsAreOK:=True
    End if
    CLOSE RESOURCE FILE($vhResFile)
End if
```

```
If(Not(<>vbFontsAreOK))
   CONFIRM("You are not using the same font set, OK?")
   If(OK=1)
      <>vbFontsAreOK:=True
   Else
      QUIT 4D
   End if
End if
```

See Also

SET STRING RESOURCE, SET TEXT RESOURCE, STRING LIST TO ARRAY.

System Variables and Sets

If the resource has been written, OK is set to 1. Otherwise, it is set to 0 (zero).

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 or Create resource file. Note that when you quit the application (or open another database), 4D automatically closes all the resource files you opened.

Example

The following example creates a resource file, adds a string resource and closes the file:

```
$vhDocRef:=Create resource file("Just a file")
If (OK=1)
    SET STRING RESOURCE(20000;"Just a string";$vhDocRef)
    CLOSE RESOURCE FILE($vhDocRef)
End if
```

See Also

Create resource file, Open resource file.

Create resource file (resFilename{; fileType{; *}}) → DocRef

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

The Create resource file command creates and opens a new resource file whose name or pathname is passed in resFileName.

If you pass a filename, the file will be located in the same folder as the structure file of the database. Pass a pathname to create a resource file located in another folder.

If the file already exists and is not currently open, Create resource file overrides it with a new empty resource file. If the file is currently open, an I/O error is returned.

If you pass an empty string in resFileName, the Save File dialog box is presented. You can then choose the location and the name of the resource file to be created. If you cancel the dialog, no resource file is created; Create resource file returns a null DocRef and sets the OK variable to 0.

If the resource file is correctly created and opened, Create resource file returns its resource file reference number and sets the OK variable to 1. If the resource file cannot be created, an error is generated.

On Macintosh, the default file type for a file created with Create resource file is “res ”.
On Windows, the default file extension is “.res”. To create a file of another type:
• On Macintosh, pass the file type in the optional parameter fileType.
• On Windows, in fileType, pass a 1- to 3-character Windows file extension or a Macintosh file type mapped using the command MAP FILE TYPES.

By default, if the * parameter is omitted, the command creates and opens the file resource fork. When this parameter is passed, the command creates and opens the file data fork (readable on both Mac OS and Windows platforms). For more information, refer to the Resources topic.

Remember to call CLOSE RESOURCE FILE for the resource file. Note, however, when you quit the application (or open another database), 4D automatically closes all the resource files you opened using Create resource file or Open resource file.

Examples

1. The following example tries to create and open on Windows, the resource file “MyPrefs.res” located in the database folder:

```
$vhResFile:=Create resource file("MyPrefs";*)
```

On Macintosh, the example tries to create and open the file “MyPrefs”.

2. The following example tries to create and open, on Windows, the resource file “MyPrefs.rsr” located in the database folder:

```
$vhResFile:=Create resource file("MyPrefs";"rsr")
```

On Macintosh, the example tries to create and open the file “MyPrefs”.

3. The following example displays the Save File dialog box:

```
$vhResFile:=Create resource file("")
If (OK=1)
    ALERT("You just created “"+Document+"”.")
    CLOSE RESOURCE FILE($vhResFile)
End if
```

See Also

CLOSE RESOURCE FILE, ON ERR CALL, Open resource file, Resources.

System Variables and Sets

If the resource file is successfully created and opened, the *OK* variable is set to 1. If the resource file could not be created or if the user clicked Cancel in the Save File dialog box, the OK variable is set to 0 (zero).

If the resource file is successfully created and opened through the Save File dialog box, the Document variable is set to the pathname of the file.

Error Handling

If the resource file could not be created or 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.

DELETE RESOURCE

DELETE RESOURCE (resType; resID{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The DELETE RESOURCE command deletes 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 resFile, the resource is searched for within the current open resource files.

If the resource does not exist, DELETE RESOURCE does nothing and sets the OK variable to 0 (zero). If the resource is found and deleted, the OK variable is set to 1.

WARNING: DO NOT delete resources that belong to 4D or to any System files. If you do so, you may provoke undesired system errors.

Examples

1. The following example deletes the resource "STR#" ID=20000:

```
   ` Note that this example will delete the first "STR#" ID=20000 resource
   ` found in any resource file currently open:
DELETE RESOURCE ("STR#";20000)
```

2. The following example deletes the resource "STR#" ID=20000 if it is found in a specified resource file:

```
    ` Note that this example will delete the resource "STR#" ID=20000
    ` only if it is present in the resource file specified by $vhResFile:
DELETE RESOURCE ("STR#";20000;$vhResFile)
    ` Note also that if there is such a resource in a currently open
    ` resource file other than that specified by $vhResFile, this resource
    ` is left untouched
```

3. The project method DELETE RESOURCES OF TYPE deletes all the resources of the type specified (as the second parameter) from the resource file specified (as the first parameter):

```
    ` DELETE RESOURCES OF TYPE Project Method
    ` DELETE RESOURCES OF TYPE ( Time ; String )
    ` DELETE RESOURCES OF TYPE ( resFile ; resType )

C_TIME($1)
C_STRING(4;$2)

RESOURCE LIST($2;$aiResID;$asResName;$1)
If(OK=1)
    For($vlElem;1;Size of array($aiResID))
        DELETE RESOURCE($2;$aiResID{$vlElem};$1)
    End for
End if
```

After this project method is present in a database, you can write:

```
    ` Delete all the resource of type "PREF" from the resource file $vhResFile
DELETE RESOURCES OF TYPE ($vhResFile;"PREF")
```

4. The project method DELETE RESOURCE BY NAME deletes a resource (of a specific type) whose name is known:

```
    ` DELETE RESOURCE BY NAME Project Method
    ` DELETE RESOURCE BY NAME ( Time ; String ; String )
    ` DELETE RESOURCE BY NAME ( resFile ; resType ; resName )

C_TIME($1)
C_STRING(4;$2)
C_STRING(255;$3)
```

```
RESOURCE LIST($2;$aiResID;$asResName;$1)
If(OK=1)
    $vlElem:=Find in array($asResName;$3)
    If($vlElem>0)
        DELETE RESOURCE($2;$aiResID{$vlElem};$1)
    End for
End if
```

After this project method is present in a database, you can write:

```
    ` Delete, from the resource file $vhResFile, the resource "PREF"
    ` whose name is “Standard Settings”:
DELETE RESOURCE BY NAME ($vhResFile;"PREF";"Standard Settings")
```

See Also

RESOURCE LIST, SET RESOURCE PROPERTIES.

System Variables and Sets

The OK variable is set to 0 if the resource does not exist. If the resource has been deleted, the OK variable is set to 1.

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.

Get component resource ID (compName; resType; originalResNum) → Number

Parameter	Type		Description
compName	String (32)	→	Component name referencing the resource
resType	String (4)	→	Resource type (4 characters), PICT or STR#
originalResNum	Number	→	Resource original number before component installation
Function result	Number	←	Current resource number

GET ICON RESOURCE

GET ICON RESOURCE (resID; resData{; fileRef})

Parameter	Type		Description
resID	Number	→	Icon resource ID number
resData	Picture	→	Picture field or variable to receive the picture
		←	Contents of the cicn resource
fileRef	Number	→	Resource file reference number, or all open resource files, if omitted

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).

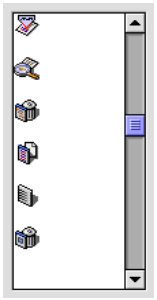
If you pass a valid resource file reference number in resFile, the resource is searched for in that file only. If you do not pass resFile, the first occurrence of the resource found in the resource files chain is returned.

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:



See Also

GET PICTURE RESOURCE.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

Get indexed string

version 11 (Modified)

Get indexed string (resID; strID{; resFile}) → String

Parameter	Type		Description
resID	Number	→	Resource ID number or 'id' attribute of the 'group' element (XLIFF)
strID	Number	→	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 command STRING LIST TO ARRAY.

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.

Example

See example for the command Month of.

See Also

Get string resource, Get text resource, STRING LIST TO ARRAY.

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	Number	→	Resource ID number
resData	Field or 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 command RESOURCE LIST.

See Also

GET ICON RESOURCE, ON ERR CALL, SET PICTURE RESOURCE.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

Error Handling

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 (resType; resID; resData{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	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 command SET RESOURCE.

See Also

BLOB Commands, Resources, SET RESOURCE.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

Error Handling

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}) → String

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	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 and sets the OK variable to 0 (zero).

Example

The following project method copies a resource, and its resource name and attributes, from one resource file to another:

```
` COPY RESOURCE Project Method
` COPY RESOURCE ( String ; Long ; Time ; Time )
` COPY RESOURCE ( resType ; resID ; srcResFile ; dstResFile )

C_STRING (4;$1)
C_LONGINT ($2)
C_TIME ($3;$4)
C_BLOB ($vxResData)
```

```
GET RESOURCE ($1;$2;$vxData;$3)
If (OK=1)
    SET RESOURCE ($1;$2;$vxData;$4)
    If (OK=1)
        SET RESOURCE NAME ($1;$2; Get resource name ($1;$2;$3);$4)
        SET RESOURCE PROPERTIES ($1;$2; Get resource properties ($1;$2;$3);$4)
    End if
End if
```

Once this project method is present in your application, you can write:

```
    ` Copy the resource 'DATA' ID = 15000 from file A to file B
COPY RESOURCE ("DATA";15000;$vhResFileA;$vhResFileB)
```

See Also

SET RESOURCE PROPERTIES.

System Variables or Sets

The OK variable is set to 0 if the resource does not exist; otherwise, it is set to 1.

Get resource properties

Get resource properties (resType; resID{; resFile}) → Number

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted
Function result	Number	←	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, Get resource properties 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. For a description of the resource attributes and their effects, please refer to the command SET RESOURCE PROPERTIES.

Example

See example for the command Get resource name.

See Also

SET RESOURCE NAME.

System Variables or 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}) → String

Parameter	Type		Description
resID	Number	→	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))
```

See Also

Get indexed string, Get text resource, SET STRING RESOURCE, STRING LIST TO ARRAY.

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}) → Text

Parameter	Type		Description
resID	Number	→	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))
```

See Also

Get indexed string, Get string resource, SET TEXT RESOURCE, STRING LIST TO ARRAY.

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}) → DocRef

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 filename, 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 Resources 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 command MAP FILE TYPES.

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 or Create 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 examples listed for the command RESOURCE TYPE LIST.

After you have opened a resource file, you can analyze the contents of the file using the commands RESOURCE TYPE LIST and RESOURCE LIST.

Examples

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”.

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”.
3. The following example displays the Open file dialog box showing all types of files:

```
$vhResFile:=Open resource file("")
```

4. The following example displays the Open file dialog box showing files created by the 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
```

See Also

CLOSE RESOURCE FILE, Create resource file, Resources.

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 Handling

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 (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 command Size of array to the array resIDs or resNames.

Examples

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
```

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
```

See Also

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.

Examples

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)
```

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.

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
```

See Also

RESOURCE LIST.

SET PICTURE RESOURCE

SET PICTURE RESOURCE (resID; resData{; resFile})

Parameter	Type		Description
resID	Number	→	Resource ID number
resData	Picture	→	New contents for the PICT resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The SET PICTURE RESOURCE command creates or rewrites the picture (“PICT”) resource whose ID is passed in resID with the picture passed in resData.

If the resource cannot be added, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top of the resource files chain (the last resource file opened).

If you pass in resData an empty picture field or variable, the command has no effect and the OK variable is set to 0.

Note: A picture resource can be several megabytes in size and even more.

See Also

GET PICTURE RESOURCE.

System Variables and Sets

If the resource has been written, OK is set to 1. Otherwise, it is set to 0 (zero).

SET RESOURCE

SET RESOURCE (resType; resID; resData{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resData	BLOB	→	New contents for the resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The SET RESOURCE command creates or rewrites the resource whose type and ID is passed in resType and resID with the data passed in the BLOB resData.

Important: You must pass a 4-character string in resType.

If the resource cannot be written, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top of the resource files chain (the last resource file opened).

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 it up to you to byte swap the data you write into the BLOB (i.e., passing Macintosh byte ordering to a command such as LONGINT TO BLOB).

Example

During a 4D session you maintain some user preferences in interprocess variables. To save these preferences from session to session, you can:

1. Use the commands SAVE VARIABLES and LOAD VARIABLES to store and retrieve the variables in variable documents on disk.

2. Use the commands VARIABLE TO BLOB, BLOB TO DOCUMENT, DOCUMENT TO BLOB and BLOB TO VARIABLE to store and retrieve the variables in BLOB documents on disk.

3. Use the commands VARIABLE TO BLOB, SET RESOURCE, GET RESOURCE and BLOB TO VARIABLE to to store and retrieve the variables in resource files on disk.
The following is an example of the third method.

In the On Exit Database Method you write:

```
    ` On Exit Database Method
If (Test path name("DB_Prefs")#Is a document)
    $vhResFile:=Create resource file("DB_Prefs")
Else
    $vhResFile:=Open resource file("DB_Prefs")
End if
If (OK=1)
    VARIABLE TO BLOB(<>vbAutoRepeat;$vxPrefData)
    VARIABLE TO BLOB(<>vlCurTable;$vxPrefData;*)
    VARIABLE TO BLOB(<>asDfltOption;$vxPrefData;*)
        ` and so on...
    SET RESOURCE("PREF";26500;$vxPrefData;$vhResFile)
    CLOSE RESOURCE FILE($vhResFile)
End if
```

In the On Startup Database Method you write:

```
    ` On Startup Database Method
C_BOOLEAN(<>vbAutoRepeat)
C_LONGINT(<>vlCurTable)
$vbDone:=False
$vhResFile:=Open resource file("DB_Prefs")
If (OK=1)
    GET RESOURCE("PREF";26500;$vxPrefData;$vhResFile)
```

```
    If (OK=1)
        $vlOffset:=0
        BLOB TO VARIABLE($vxPrefData;<>vbAutoRepeat;$vlOffset)
        BLOB TO VARIABLE($vxPrefData;<>vlCurTable;$vlOffset)
        BLOB TO VARIABLE($vxPrefData;<>asDfltOption;$vlOffset)
            ` and so on...
        $vbDone:=True
    End if
    CLOSE RESOURCE FILE($vhResFile)
End if
If(Not($vbDone))
    <>vbAutoRepeat:=False
    <>vlCurTable:=0
    ARRAY STRING(127;<>asDfltOption;0)
End if
```

See Also

BLOB Commands, GET RESOURCE.

System Variables and Sets

If the resource is written, OK is set to 1. Otherwise, it is set to 0 (zero).

SET RESOURCE NAME

SET RESOURCE NAME (resType; resID; resName{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resName	String	→	New name for the resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The SET RESOURCE NAME command changes 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, SET RESOURCE NAME does nothing and sets the OK variable to 0 (zero).

WARNING: DO NOT change the names of resources that belong to 4D or to any System files. If you do so, you may provoke undesired system errors.

Note: Resource names can be up to 255 characters in length. They are not case sensitive, but are diacritical sensitive.

Example

See example for the command Get resource name.

See Also

SET RESOURCE PROPERTIES.

System Variables or Sets

The OK variable is set to 0 if the resource does not exist, otherwise it is set to 1.

SET RESOURCE PROPERTIES

SET RESOURCE PROPERTIES (resType; resID; resAttr{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resAttr	Number	→	New attributes for the resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The SET RESOURCE PROPERTIES command changes 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, SET RESOURCE PROPERTIES does nothing and sets the OK variable to 0 (zero).

The numeric value you pass in resAttr must be seen as a bit field value whose bits have special meaning. The following predefined constants are provided by 4D:

Constant	Type	Value
System heap resource mask	Long Integer	64
System heap resource bit	Long Integer	6
Purgeable resource mask	Long Integer	32
Purgeable resource bit	Long Integer	5
Locked resource mask	Long Integer	16
Locked resource bit	Long Integer	4
Protected resource mask	Long Integer	8
Protected resource bit	Long Integer	3
Preloaded resource mask	Long Integer	4
Preloaded resource bit	Long Integer	2
Changed resource mask	Long Integer	2
Changed resource bit	Long Integer	1

Using these constants, you can build any resource attributes value. See examples below.

Resource Attributes and Their Effects

• System heap
If this attribute is set, the resource will be loaded into the system memory rather than into 4D memory. You should not use this attribute, unless you **really** know what you are doing.

• Purgeable
If this attribute is set, after the resource has been loaded, you can purge it from memory if space is required for allocation of other data. Since you load resources into 4D BLOBs, it is a good idea to have all your own resources purgeable in order to reduce memory usage. However, if you frequently access this resource during a working session, you might want to make it non-purgeable in order to reduce disk access due to frequent reloading of a purged resource.

• Locked
If this attribute is set, you will not be able to relocate the resource (unmovable) after it is loaded into memory. A locked resource cannot be purged even if it is purgeable. Locking a resource has the undesirable effect of fragmenting the memory space. DO NOT use this attribute, unless you **really** know what you are doing.

• Protected
If this attribute is set, you can no longer change the name, ID number or the contents of a the resource. You can no longer delete this resource. However, you can call SET RESOURCE PROPERTIES to clear this attribute; then you can again modify or delete the resource. Most of the time, you will not use this attribute. Note: This attribute has no effect on Windows.

• Preloaded
If this attribute is set, the resource is automatically loaded into memory if the resource file where it is located is open. This attribute is useful for optimizing resource loading when a resource file is opened. Most of the time, you will not use this attribute.

• Changed
If this attribute is set, the resource is marked as “must be saved on disk” when the resource file where it is located is closed. Since the 4D command SET RESOURCE handles the writing and rewriting of resources internally, you should not use this attribute, unless you **really** know what you are doing.

You will usually use the attribute purgeable and, more rarely, Preloaded and Protected.

WARNING: DO NOT change the attributes of resources that belong to 4D or to any System files. If you do so, you may provoke undesired system errors.

Examples

1. See example for the command Get resource name.

2. The following example makes the resource 'STR#' ID=17000 purgeable, but leaves the other attributes unchanged:

```
$vlResAttr:=Get resource properties ('STR#';17000;$vhResFile)
SET RESOURCE PROPERTIES('STR#';17000;$vlResAttr ?+ Purgeable resource bit;
                                                    $vhMyResFile)
```

3. The following example makes the resource 'STR#' ID=17000 preloaded and non purgeable:

```
SET RESOURCE PROPERTIES('STR#';17000;Preloaded resource mask;$vhResFile)
```

4. The following example makes the resource 'STR#' ID=17000 preloaded but purgeable:

```
SET RESOURCE PROPERTIES('STR#';17000;Preloaded resource mask+
                                    Purgeable resource mask;$vhResFile)
```

See Also

SET RESOURCE NAME.

System Variables or Sets

The OK variable is set to 0 if the resource does not exist; otherwise, it is set to 1.

SET STRING RESOURCE

SET STRING RESOURCE (resID; resData{; resFile})

Parameter	Type		Description
resID	Number	→	Resource ID number
resData	String	→	New contents for the STR resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The SET STRING RESOURCE command creates or rewrites the string (“STR ”) resource whose ID is passed in resID with the string passed in resData.

If the resource cannot be added, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top the resource files chain (the last resource file opened).

Note: A string resource can contain up to 255 characters.

See Also

Get string resource, SET TEXT RESOURCE.

System Variables and Sets

If the resource has been written, OK is set to 1. Otherwise, it is set to 0 (zero).

SET TEXT RESOURCE

SET TEXT RESOURCE (resID; resData{; resFile})

Parameter	Type		Description
resID	Number	→	Resource ID number
resData	String	→	New contents for the TEXT resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The SET TEXT RESOURCE command creates or rewrites the text (“TEXT”) resource whose ID is passed in resID with the text or string passed in resData.

If the resource cannot be added, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top the resource files chain (the last resource file opened).

Note: A text resource can contain up to 32,000 characters.

See Also

Get text resource, SET STRING RESOURCE.

System Variables and Sets

If the resource has been written, OK is set to 1. Otherwise, it is set to 0 (zero).

STRING LIST TO ARRAY

version 11 (Modified)

STRING LIST TO ARRAY (resID; strings{; resFile})

Parameter	Type		Description
resID	Number	→	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.

Example

See example for the command ARRAY TO STRING LIST.

See Also

ARRAY TO STRING LIST, Get indexed string, Get string resource, Get text resource.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

45

Secured Protocol

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 section Using SSL Protocol.

The certificate request uses keypairs generated with the command GENERATE ENCRYPTION KEYPAIR 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 command GENERATE ENCRYPTION KEYPAIR.

Pass in certifRequest an empty BLOB. Once the command has been executed, it contains the certificate request at the PKCS format. You can store this request in a text file, 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.

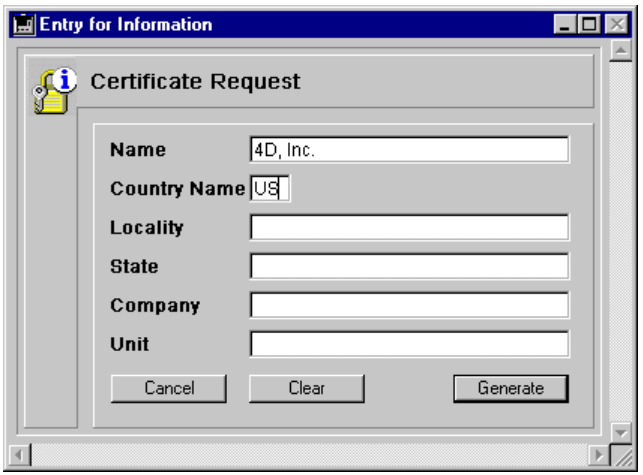
The required codes and names may change according to the certificate authority and the certificate use. However, within a normal use of the certificate (Web server connections via SSL), the arrays should contain the following items:

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
```

See Also

GENERATE ENCRYPTION KEYPAIR, Using SSL Protocol.

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...1024] 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 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 SSL 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 1024 bits keys to increase the encryption security but the Web application connections will be slowed down.

Notes:
• If you generate keys in order to establish a SSL certificate request, pay attention to the fact that only 512 bits and 1024 bits key length are admitted.
• Many browsers will not accept keys with a length greater than 512 bits. Additionally, the "Export" version of the encryption system library which is provided by default by 4D, Inc., does not provide support for key lengths greater than 512 bits. For more information, please refer to the section Using SSL Protocol).

This command will generate keys at the PKCS format, 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 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 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 SSL 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 two commands ENCRYPT BLOB and DECRYPT BLOB second syntax.

Example

See example for command ENCRYPT BLOB.

See Also

DECRYPT BLOB, ENCRYPT BLOB, GENERATE CERTIFICATE REQUEST.

46

Selection

ALL RECORDS

Selection

version 3

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
```

See Also

DISPLAY SELECTION, MODIFY SELECTION, ORDER BY, QUERY, Records in selection, Records in table.

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.

Examples

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))
```

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
```

3. This example uses a method:

```
ALL RECORDS ([Employees])
APPLY TO SELECTION([Employees];M_Cap)
```

System Variables or 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.

See Also

EDIT FORMULA, Sets.

Before selection

version 3

Before selection {(table)} → Boolean

Parameter	Type		Description
table	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 LAST RECORD, FIRST 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
```

See Also

End selection, FIRST RECORD, Form event, PREVIOUS RECORD, PRINT SELECTION.

DELETE SELECTION {(table)}

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.

The Completely Delete option in the Table Properties dialog box allows you to increase the speed of deletions when DELETE SELECTION is used.

Examples

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
OUTPUT FORM ([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
```

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
```

See Also

DISPLAY SELECTION, MODIFY SELECTION, Record Locking, Sets.

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), 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:
• if you pass No Selection (0), it will not be possible to select a record in the list.
• if you pass Single Selection (1), only one record can be selected at a time.

• if you pass Multiple Selection (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.
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 SET ENTERABLE command can be used to enable or disable the Enter in list mode on the fly.

Some rules regarding the optional * parameter:
- 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.

Examples

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
```

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.

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.

See Also

Form event, MODIFY SELECTION, Sets.

Displayed line number

Selection

version 11 (Modified)

Displayed line number → Longint

Parameter	Type		Description
This command does not require any parameters			
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.

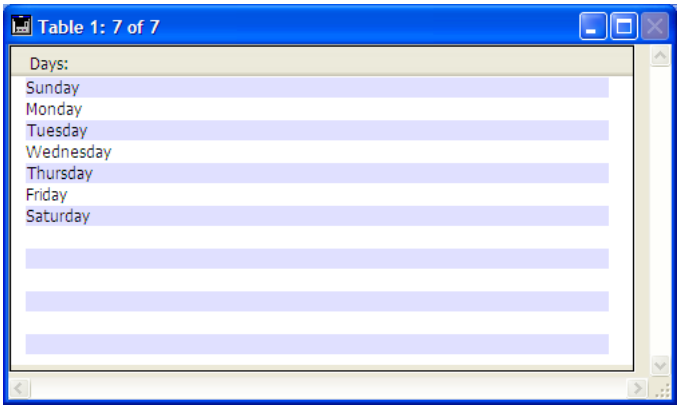
In the case of a list of records, when the displayed row is not empty (when it is linked to a record), the value returned by Displayed line number is identical to the value returned by Selected record number.

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
        SET RGB COLORS([Table 1]Field1; -1; 0x00FFFFFF)
    Else
            `Black on light blue for odd row text
        SET RGB COLORS([Table 1]Field1; -1; 0x00E0E0FF)
    End if
End if
```



See Also

Form event, Selected record number.

End selection

Selection

version 3

End selection {(aTable)} → Boolean

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 LAST RECORD, FIRST 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
```

See Also

Before selection, Form event, LAST RECORD, NEXT RECORD, PRINT SELECTION.

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])
```

See Also

Before selection, End selection, LAST RECORD, NEXT RECORD, PREVIOUS RECORD.

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")
```

See Also

HIGHLIGHT RECORDS.

System Variables or Sets

If the command was executed properly, the system variable OK is set to 1. Otherwise, it is set to 0.

GOTO SELECTED RECORD ({table; }record)

Parameter	Type		Description
aTable	Table	→	Table in which to go to the selected record, or Default table, if omitted
record	Number	→	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.

If there are no records in the current selection, or the number is not in the selection, then GOTO SELECTED RECORD does nothing.

If you pass 0 in position, 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
```

See Also

Selected record number.

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
*	*	→	Disable the automatic scroll of the list

Description

The HIGHLIGHT RECORDS command allows you to highlight 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 will be applied to the records in that set for the table defined.
• If you omit the setName parameter, the command will only highlight the records in the current UserSet set. This set is only managed in Design mode and when calling the MODIFY SELECTION / DISPLAY 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, causes the disabling of the automatic scroll function of the list if the highlighted records are not visible. This mechanism authorizes customized scroll management using the new SCROLL LINES 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 should 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.

See Also

GET HIGHLIGHTED RECORDS, SCROLL LINES.

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, LAST RECORD has no effect.

Example

The following example makes the last record of the [People] table the current record:

```
LAST RECORD ([People])
```

See Also

Before selection, End selection, FIRST RECORD, NEXT RECORD, PREVIOUS RECORD.

MODIFY SELECTION

Selection

version 2004 (Modified)

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.

See Also

DISPLAY SELECTION, Form event, Sets.

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.

See Also

Before selection, End selection, FIRST RECORD, LAST RECORD, PREVIOUS RECORD.

ONE RECORD SELECT

Selection

version 3

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.

Historical 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. In version 6, 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

version 3

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, Before selection returns TRUE, and there is no current record. If Before selection returns TRUE, use FIRST RECORD, LAST RECORD, or GOTO SELECTED RECORD to move the current record pointer back into the current selection.

See Also

Before selection, End selection, FIRST RECORD, LAST RECORD, NEXT RECORD.

Records in selection

version 3

Records in selection {(aTable)} → Number

Parameter	Type		Description
aTable	Table	→	Table for which to return number of selected records, or Default table, if omitted
Function result	Number	←	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
```

See Also

Records in table.

REDUCE SELECTION ({aTable; }number)

Parameter	Type		Description
aTable	Table	→	Table for which to reduce the selection, or Default table, if omitted
number	Number	→	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(0) is executed, there is no longer any selection nor any current records in the aTable.

Examples

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
      ` ...who sold for more than $50,000
   QUERY([Dealers];[Dealers]Country=[Countries]Name;*)
   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
```

See Also

ORDER BY, QUERY, SCAN INDEX, Sets.

SCAN INDEX (aField; number{; > or <})

Parameter	Type		Description
aField	Field	→	Indexed field on which to scan index
number	Number	→	Number of records to return
> or <		→	> 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, but there is no current record.

If you specify more records than exist in the table, SCAN INDEX will return all the records.

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
OUTPUT FORM([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
OUTPUT FORM([Customers];"Thanks Letter")
PRINT SELECTION([Customers]) ` Print the letters
```

See Also

ORDER BY, QUERY, REDUCE SELECTION.

Selected record number

Selection

version 3

Selected record number {(aTable)} → Number

Parameter	Type		Description
aTable	Table	→	Table for which to return the selected record number or Default table, if omitted
Function result	Number	←	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
```

See Also

About Record Numbers, GOTO SELECTED RECORD, Records in selection.

47

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. UserSet and LockedSet are process sets. 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 all the processes of every client as well as to the server process (stored procedures).
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. Version 6 introduces local/client sets. The name of a local/client set is preceded by the dollar sign ($). Unlike other types of sets, a local/client set is stored on the client machine.

Note: For more information about the use of sets in client/server mode, please refer to the 4D Server and Sets section if the *4D Server Reference* Manual.

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.

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.
OUTPUT FORM ([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, ARRAY TO SELECTION and DELETE SELECTION commands create a set named LockedSet when used in a multi-processing environment. LockedSet indicates which records were locked during the execution of the command.

See Also

Identifiers.

ADD TO SET

version 3

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.

See Also

REMOVE FROM SET.

CLEAR SET

version 3

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. CLEAR SET 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.

See Also

CREATE EMPTY SET, CREATE SET, LOAD SET.

COPY SET

version 6.0

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.

Both sets can be process, interprocess or local sets.

4D Server: In Client/Server, interprocess and process sets are maintained on the server machine, while local sets are maintained on the client machines. COPY SET allows you to copy sets between the two machines. See the discussion 4D Server and Sets in the *4D Server Reference* manual for more information.

Examples

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")
```

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")
```

See Also

Sets.

USE SET

version 3

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 the 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 INTERSECTION, UNION, 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
```

See Also

CLEAR SET, LOAD SET.

CREATE EMPTY SET

version 3

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.

See Also

CLEAR SET, CREATE SET.

CREATE SET

version 3

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
```

See Also

CLEAR SET, CREATE EMPTY SET, Identifiers.

CREATE SET FROM ARRAY

version 6.7 (Modified)

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.

See Also

BOOLEAN ARRAY FROM SET, CREATE SELECTION FROM ARRAY, Identifiers.

DIFFERENCE (set; subtractSet; resultSet)

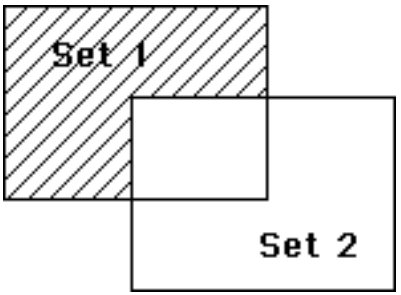
Parameter	Type		Description
set	String	→	Set
subtractSet	String	→	Set to subtract
resultSet	String	→	Resulting set

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, interprocess and process sets are maintained on the server machine, while local sets are maintained on the client machines. DIFFERENCE requires the three sets to be on the same machine. Consequently, all or none of the sets must be local. See the discussion 4D Server and Sets 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
```

See Also

INTERSECTION, UNION.

INTERSECTION (set1; set2; resultSet)

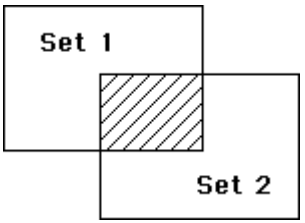
Parameter	Type		Description
set1	String	→	First set
set2	String	→	Second set
resultSet	String	→	Resulting set

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.

4D Server: In Client/Server, interprocess and process sets are maintained on the server machine, while local sets are maintained on the client machines. INTERSECTION requires the three sets to be on the same machine. Consequently, all or none of the sets must be local. See the discussion 4D Server and Sets 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
```

See Also

DIFFERENCE, UNION.

Is in set

version 3

Is in set (set) → Boolean

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
```

See Also

ADD TO SET, REMOVE FROM 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 or 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.

See Also

SAVE SET.

Records in set

version 3

Records in set (set) → Number

Parameter	Type		Description
set	String	→	Name of the set to test
Function result	Number	←	Number of records in test

Description

Records in set returns the number of records in set. If set does not exist, or if there are no records in set, Records in set 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.")
```

See Also

Records in selection, Records in table.

REMOVE FROM SET

version 6.0

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, REMOVE FROM SET has no effect.

See Also

ADD TO 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 need not 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 invalidate 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 or 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.

See Also

LOAD SET.

UNION (set1; set2; resultSet)

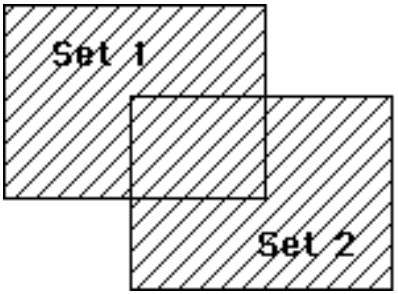
Parameter	Type		Description
set1	String	→	First set
set2	String	→	Second set
resultSet	String	→	Resulting set

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, interprocess and process sets are maintained on the server machine, while local sets are maintained on the client machines. UNION requires the three sets to be on the same machine. Consequently, all or none of the sets must be local. See the discussion 4D Server and Sets 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
```

See Also

DIFFERENCE, INTERSECTION.

48

SQL

Beginning with version 11, 4D includes an integrated SQL kernel. The program also includes an SQL server that third-party applications can query (via the 4D OBDC driver). The different ways of accessing the 4D SQL kernel, the configuration of the SQL server and the description of SQL queries are detailed in a separate manual, the *4D SQL Reference* manual.

The "SQL" theme groups together various 4D commands concerning the use of SQL in 4D:
• Control of the SQL server: START SQL SERVER and STOP SQL SERVER
• Use of the integrated SQL kernel: SET FIELD VALUE NULL, Is field value Null, QUERY BY SQL, GET LAST SQL ERROR
• Direct access to SQL external data sources (SQL pass-through): GET DATA SOURCE LIST, Get current data source, Begin SQL, End SQL, USE INTERNAL DATABASE, USE EXTERNAL DATABASE.

Begin SQL

Parameter	Type	Description

This command does not require any parameters

Description

Begin SQL is a keyword that can be used in the Method editor to indicate the beginning of a sequence of commands that must be interpreted by the integrated SQL engine of 4D.

A sequence of SQL commands started with Begin SQL must be closed with the End 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.

See Also

End SQL.

End SQL

End SQL

Parameter	Type	Description
This command does not require any parameters		

Description

End SQL is a keyword indicating the end of a sequence of commands in the Method editor that must be interpreted by the integrated SQL engine of 4D.

A sequence of SQL statements must be surrounded by the Begin SQL and End SQL keywords. For more information, please refer to the description of the Begin SQL keyword.

See Also

Begin SQL.

Get current data source

Get current data source → String

Parameter	Type		Description
This command does not require any parameters			
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 ("External data source" theme).
This command lets you check the current data source, generally before executing an SQL query.

See Also

GET DATA SOURCE LIST, USE EXTERNAL DATABASE, USE INTERNAL DATABASE.

GET DATA SOURCE LIST

version 11

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 USE EXTERNAL DATABASE 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 USE INTERNAL DATABASE 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 "External data source" theme:

Constant	Type	Value
User Data Source	Longint	1
System Data Source	Longint	2

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.

Example

Example using a user data source:

```
ARRAY TEXT(arrDSN;0)
ARRAY TEXT(arrDSNDrivers;0)
GET DATA SOURCE LIST(User Data Source;arrDSN;arrDSNDrivers)
```

See Also

Get current data source, USE EXTERNAL DATABASE, USE INTERNAL DATABASE.

System Variables or 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.

GET LAST SQL ERROR

version 11

GET LAST SQL ERROR (codesArray; intCompArray; textArray)

Parameter	Type		Description
codesArray	Number array	←	Error numbers
intCompArray	String array	←	Internal component codes
textArray	String array	←	Text of errors

Description

The GET LAST SQL ERROR command returns information about the current stack of errors concerning the use of the SQL kernel.

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 SQL Engine Errors section.

Is field value Null

Is field value Null (aField) → Boolean

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, please refer to the *4D SQL Reference* manual.

See Also

SET FIELD VALUE NULL.

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.

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, please refer to the *4D SQL Reference* manuel.

The sqlFormula parameter can use references to 4D expressions. The syntax to use is the same as for the integrated ODBC commands or the code included between the Begin SQL/End SQL tags, i.e.: <<MyVar>> or :MyVar.
For more information, refer to the External Data Source Commands section.

Note: This command is compatible with the SET QUERY LIMIT and SET QUERY DESTINATION 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.

Examples
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)
```

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)
```

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)
```

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)
```

See Also

QUERY BY FORMULA.

System Variables or 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 (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 kernal of 4D. For more information, please refer to the *4D SQL Reference* manual.

Note: It is possible to disallow the Null value for 4D fields at the Structure editor level (see the *Design Reference* manual).

See Also

Is field value Null.

START SQL SERVER

Parameter	Type	Description
This command 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.

See Also

STOP SQL SERVER.

System Variables or 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

version 11

STOP SQL SERVER

Parameter	Type	Description
This command 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.

See Also

START SQL SERVER.

USE INTERNAL DATABASE

Parameter	Type	Description
This command does not require any parameters		

Description

The USE INTERNAL DATABASE command is used to close any external connections opened using the USE EXTERNAL DATABASE command and to reconnect the application to the local 4D database. Any SQL statements executed within Begin SQL/End SQL structures will then be sent to the local 4D database.

This command does nothing if no external connection has been opened using the USE EXTERNAL DATABASE command.

See Also

Get current data source, GET DATA SOURCE LIST, USE EXTERNAL DATABASE.

System Variables or 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.

USE EXTERNAL DATABASE

version 11

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

Description

The USE EXTERNAL DATABASE command establishes a connection between the 4D application and the data source designated by the sourceName parameter. You can get a list of data sources available on the machine using the GET DATA SOURCE LIST command.

Once the connection is established, all SQL statements executed within Begin SQL/End SQL structures from then on will be sent to this source (SQL pass-through) in the current process, until the USE INTERNAL DATABASE command or another USE EXTERNAL DATABASE statement is executed.

Pass any ID information required by the data source in the user and password parameters.

Example

```
C_TEXT(sourceNamesArr;sourceDriversArr;0)
GET DATA SOURCE LIST(1;sourceNamesArr;sourceDriversArr)  `User data sources
If (Find in array(sourceNamesArr;"emp")#-1)  `If the source emp does exist
    USE EXTERNAL DATABASE("emp";"tiger";"scott")
    Begin SQL
    ... `SQL statements
    End SQL
End if
```

See Also

Get current data source, GET DATA SOURCE LIST, USE INTERNAL DATABASE.

System Variables or 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.

49

String

Overview

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).

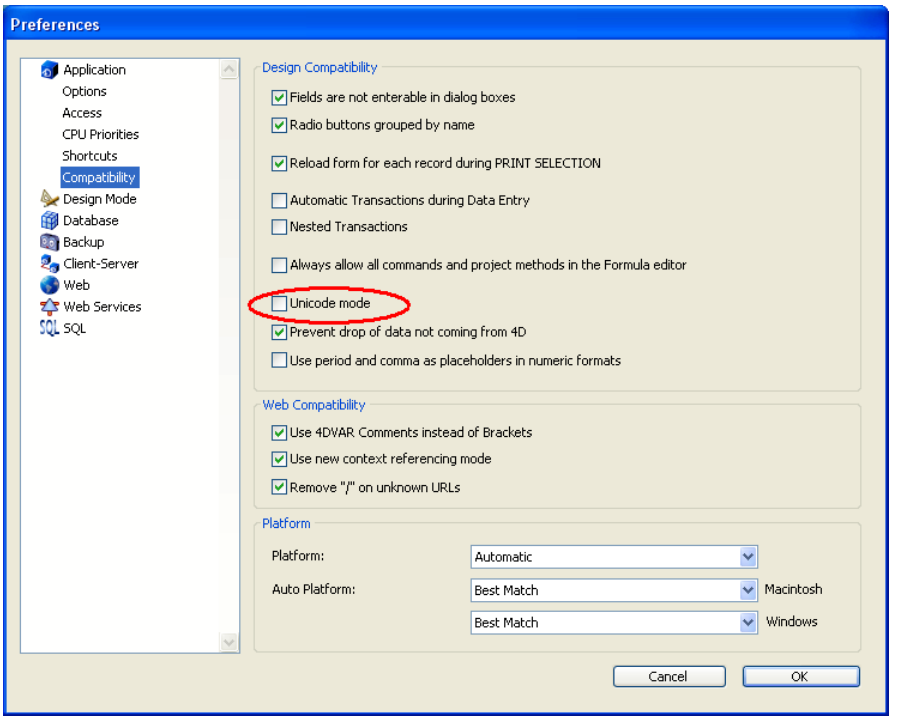
Warning: In Unicode in 4D v11, the following character codes are reserved and must never be included in a text:

0
65534 (FFFE)
65535 (FFFF)

ASCII Compatibility Mode

Previous versions of 4D worked with the extended ASCII table of Macintosh (see the ASCII Codes section). By default, databases converted from a previous version of 4D continue to function in this mode called "ASCII compatibility mode".

It is possible to apply the Unicode mode to converted databases via the Unicode Mode selector of the Get database parameter and SET DATABASE PARAMETER commands or via the **Unicode Mode** option found on the Application/Compatibility page of the Preferences:



Note: This mode is specific to each database. It is therefore possible to have a Unicode database coexisting with non-Unicode components (or vice versa).

In most cases, the initial functioning of applications is not affected by this setting, since 4D handles any necessary character conversions internally. Moreoever, the most common characters (a-z, 0-9, and so on) have the same value (from1 to 127) in both Unicode and ASCII (Windows and Mac OS).
However, certain language statements, more particularly those using commands that work with character strings, may require some adaptation. For example, the statement Char(200) will not return the same value in Unicode as in ASCII. This manual describes the differences in functioning between the Unicode mode and the ASCII compatibility mode for each command concerned.

Introduction

The character reference symbols:

```
[[...]]  On Windows
‹...› or [[...]] On Macintosh
```

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.

Note: On Macintosh, you obtain the first two symbols by typing Option+"<" and Option+">".

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
```

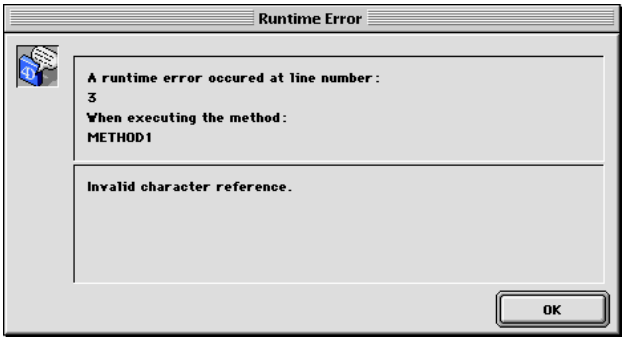
Advanced note about invalid character reference

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:

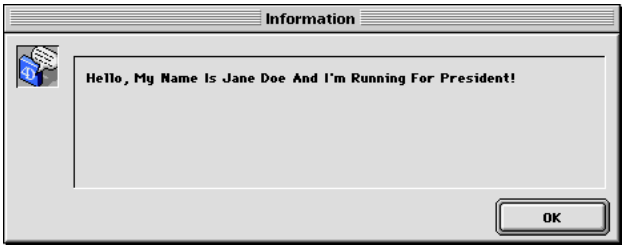
```
   ` Capitalize text project method
   ` Capitalize text ( Text ) -> Text
   ` Capitalize text ( Source text ) -> Capitalized text
```

```
$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:



See Also

ASCII Codes, Char, Character code.

Change string

String

version 3

Change string (source; newChars; where) → String

Parameter	Type		Description
source	String	→	Original string
newChars	String	→	New characters
where	Number	→	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. Change string 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"
```

See Also

Delete string, Insert string, Replace string.

Char

String

version 11 (Modified)

Char (charCode) → String

Parameter	Type		Description
charCode	Number	→	Character code
Function result	String	←	Character represented by the charCode

Description

The Char command returns the character whose code is charCode.

• If the database operates in Unicode mode (default mode for databases created starting with 4D version 11), you must pass a UTF-16 value (included between 1 and 65535) in charCode.
• If the database operates in ASCII compatibility mode, you must pass an ASCII code (included between 0 and 255) in charCode.
For more information about the different modes for managing strings in 4D, please refer to the About Unicode section.

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.

Important: In ASCII compatibility mode, all the text values, fields, or variables that you have not yet converted to another ASCII map are Mac OS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

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.")
```

See Also

About Unicode, ASCII Codes, Character code, Character Reference Symbols.

Character code

String

version 11 (Modified)

Character code (character) → Number

Parameter	Type		Description
character	String	→	Character for which you want to get the code
Function result	Number	←	Character code

Description

The Character code command returns the current character code of character.
• If the database is operating in Unicode mode (default mode for databases created with version 11 and higher of 4D), the command returns the Unicode UTF-16 code of character (included between 1 and 65535).
• If the database is operating in ASCII compatibility mode, the command returns the ASCII code of character (included between 0 and 255).
For more information about the different modes for managing strings in 4D, please refer to the About Unicode section.

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 a UTF-16 or ASCII code represents.

Important: In ASCII compatibility mode, all the text values, fields, or variables that you have not yet converted to another ASCII map are Mac OS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

Examples

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"))
```

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
```

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.

See Also

ASCII Codes, Char, Character Reference Symbols.

Preliminary note

This command meets very specific needs related to non-Roman character sets. Its use is very limited.

Convert case (string; target; srcMask) → String

Parameter	Type		Description
string	String	→	String to be converted
target	Integer	→	Type of conversion
srcMask	Longint	→	Part to be converted
Function result	String	←	Converted string

Description

The Convert case command calls the TransliterateText function of the Apple Script Manager directly, which permits the conversion of text into different subvariants on non-Roman systems. For a complete description of this function, please refer to the following address: http://developer.apple.com/documentation/mac/Text/Text-402.html

The parameters of this command correspond to those of the TransliterateText function. Note that 4D uses the smSystemScript scriptCode.

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. The following is a (non-exhaustive) list of these identifiers:

Identifier	IANA Name
1	UTF-16BE
2	UTF-16LE
7	UTF-8
8	UTF-7
9	US-ASCII
10	ebcdic-cp-us
100	x-mac-roman
101	windows-1252
102	x-mac-ce
103	windows-1250
104	x-mac-cyrillic
105	windows-1251
106	x-mac-greek
107	windows-1253
108	x-mac-turkish
109	windows-1254
110	x-mac-arabic
111	windows-1256
112	x-mac-hebrew
113	windows-1255
114	x-mac-ce
115	windows-1257

1000	Shift_JIS
1001	ISO-2022-JP
1002	Big5
1003	EUC-KR
1004	KOI8-R
1005	ISO-8859-1
1006	ISO-8859-2
1007	ISO-8859-3
1008	ISO-8859-4
1009	ISO-8859-5
1010	ISO-8859-6
1011	ISO-8859-7
1012	ISO-8859-8
1013	ISO-8859-9

The standard character set names are specified by the IANA. The command accepts the main name of the character set as well as all its referenced aliases (for example, “IO-8859-1” can be named “CP819,” “csISOLatin1,” “latin1,” or yet again “l1”). 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.

Note: This command only works when 4D is executed in Unicode mode (the Unicode option must be checked in the 4D Preferences, see the "About Unicode" section). If it is used in compatibility mode (non-Unicode), convertedBLOB is returned empty and the OK variable is set to 0.

See Also

Convert to text.

System Variables or Sets

If the command has been correctly executed, the OK variable is set to 1. Otherwise, it is set to 0.

Convert to text (BLOB; charSet) → Text

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. In version 11, 4D uses the Unicode character set by default.

In charSet, pass the character set of the text contained in blob, which will be used for the conversion. 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.

Note: This command only works when 4D is executed in Unicode mode (the Unicode option must be checked in the 4D Preferences, see the “About Unicode” section). If it is used in compatibility mode (non-Unicode), it returns an empty string and the OK variable is set to 0.

See Also

CONVERT FROM TEXT.

System Variables or Sets

If the command has been correctly executed, the OK variable is set to 1. Otherwise, it is set to 0.

Delete string

version 3

Delete string (source; where; numChars) → String

Parameter	Type		Description
source	String	→	String from which to delete characters
where	Number	→	First character to delete
numChars	Number	→	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 gets the first two characters of vtOtherVar
vtResult:=Delete string(vtOtherVar;3;32000)
```

See Also

Change string, Insert string, Replace string.

Get localized string (resName) → String

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 current 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.

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”.

See Also

Get current database localization.

System Variables or 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.

Insert string (source; what; where) → String

Parameter	Type		Description
source	String	→	String in which to insert the other string
what	String	→	String to insert
where	Number	→	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"
```

See Also

Change string, Delete string, Replace string.

ISO to Mac

String

version 11 (Modified)

ISO to Mac (text) → String

Parameter	Type		Description
text	String	→	Text expressed using standard Web character set
Function result	String	←	Text expressed using Mac OS ASCII map

Compatibility Note: This command only works when the database is executed in ASCII compatibility mode. In Unicode mode, it does nothing (the text string is returned without modification). Beginning with version 11 of 4D, this command is thus 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.

Description

The ISO to Mac command returns text, expressed using the Mac OS ASCII map, equivalent to the text you pass in text, expressed using the ISO Latin-1 character map.

You will generally not need to use this command.

This command expects a text parameter expressed using the ISO Latin-1 character map.

4D converts characters received from and sent to a Web Browser. As a result, the text values you deal with, inside a Web Connection process, are always expressed using the Mac OS ASCII map.

Generally, when running on Windows, you do not need to convert ASCII codes. In ASCII compatibility mode (non-Unicode), when you copy or paste text between 4D and Windows or when you import or export data, 4D automatically performs these conversions. However, when you use disk read/write commands such as SEND PACKET or RECEIVE PACKET, 4D does not perform any ASCII code conversion.

Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are Mac OS-encoded on both Macintosh and Windows. For more information, see the ASCII Codes section.

On Windows, it is necessary that, in this case, you do not filter the characters using an output filter ASCII map.

Consequently, no matter what the platform, if you want to use RECEIVE PACKET to read ISO Latin-1 HTML documents from the disk, you just need to convert the text using ISO to Mac. This is the main purpose of the ISO to Mac command.

Example

The following line of code converts the (assumed) ISO Latin-1 encoded text stored in vtSomeText into a Mac OS encoded text:

```
   ` Read some text from an ISO Latin-1 HTML document
RECEIVE PACKET ($vhDocRef;vtSomeText;16*1024)
vtSomeText:=ISO to Mac(vtSomeText)
```

See Also

ASCII Codes, Mac to ISO, RECEIVE PACKET, USE CHARACTER SET.

Length

version 3

Length (aString) → Number

Parameter	Type		Description
aString	String	→	String for which to return length
Function result	Number	←	Length of string

Description

Length is used to find the length of aString. Length returns the number of characters that are in aString.

Note: The test If (vtAnyText="") is equivalent to the test If(Length(vtAnyText)=0).

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 (aString; *) → String

Parameter	Type		Description
aString	String	→	String to convert to lowercase
*	*	→	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.

Examples

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
```

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"
```

See Also

Uppercase.

Mac to ISO

String

version 11 (Modified)

Mac to ISO (text) → String

Parameter	Type		Description
text	String	→	Text expressed using Mac OS ASCII map
Function result	String	←	Text expressed using standard Web character set

Compatibility Note: This command only works when the database is executed in ASCII compatibility mode. In Unicode mode, it does nothing (the text string is returned without modification). Beginning with version 11 of 4D, this command is thus 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.

Description

The Mac to ISO command returns text equivalent to that passed in text, but expressed using the Web characters table found in the **Standard Set** menu of the Web/Options page in the application Preferences.

You will generally not need to use this command.

This command expects a text parameter expressed using the Mac OS ASCII map.

4D converts characters received from and sent to a Web Browser. As a result, the text values you deal with, inside a Web Connection process, are always expressed using the Mac OS ASCII map.

Generally, when running on Windows, you do not need to convert ASCII codes. In ASCII compatibility mode (non-Unicode), when you copy or paste text between 4D and Windows or when you import or export data, 4D automatically performs these conversions. However, when you use disk read/write commands such as SEND PACKET or RECEIVE PACKET, you need to explicitly invoke ASCII conversions.

Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are Mac OS-encoded on both Macintosh and Windows. For more information, see the ASCII Codes section.

On Windows, it is necessary that, in this case, you do not filter the characters using an output filter ASCII map.

Consequently, no matter what the platform, if you want to write ISO Latin-1 HTML documents or documents using other Web character sets on disk, you just need to convert the text using Mac to ISO. This is the main purpose of the Mac to ISO command.

Examples

1. The following line of code converts by default the (assumed) Mac OS encoded text stored in vtSomeText into an ISO-Latin 1 encoded text:

```
vtSomeText:=Mac to ISO(vtSomeText)
```

2. While developing a 4D Web Server application, you build HTML documents that you later send over Intranet or Internet, using the SEND HTML FILE command. Some of these documents have references or links to other documents.

The following project method calculates an HTML-based pathname from the Windows or Macintosh pathname received as parameter:

```
` HTML Pathname project method
` HTML Pathname ( Text ) -> Text
` HTML Pathname ( Native File Manager Pathname ) -> HTML Pathname

C_TEXT($0;$1)
C_LONGINT($vlChar;$vlAscii)
C_STRING(31;$vsChar)

$0:=""
If (On Windows )
   $1:=Replace string($1;"\";"/")
Else
   $1:=Replace string($1;":";"/")
End if
$1:=Mac to ISO($1)
For ($vlChar;1;Length($1))
   $vlAscii:=Character code($1[[$vlChar]])
   Case of
      : ($vlAscii>=127)
         $vsChar:="%"+Substring(String($vlAscii;"&$");2)
```

```
        : (Position(Char($vlAscii);":<>&%= "+Char(34))>0)
            $vsChar:="%"+Substring(String($vlAscii;"&$");2)
    Else
        $vsChar:=Char($vlAscii)
    End case
    $0:=$0+$vsChar
End for
```

Note: The On Windows project method is listed in the System Documents section.

Once this project method is present in your database, if you want to include a list of FTP links to documents present in a particular directory, you can write:

```
    ` Interprocess variables set, for instance, in the On Startup database method
<>vsFTPURL:="ftp://123.4.56.78/Spiders/"
<>vsFTPDirectory:="APS500:Spiders:" ` Here, a Mac OS File Manager pathname
    ` ...

    ` ...
ARRAY STRING(31;$asDocuments;0)
DOCUMENT LIST(...;$asDocuments)
$vlNbDocuments:=Size of array($asDocuments)
jsHandler:=...
For ($vlDocument;1;$vlNbDocuments)
    vtHTMLCode:=vtHTMLCode+"<P><A HREF="+Char(34)+<>vsFTPURL+HTML Pathname
            (Substring($1+$asDocuments{$vlDocument};Length(<>vsFTPDirectory)+1))+
                Char(34)+jsHandler+"> "+$asDocuments{$vlDocument}+
                    "</A></P>"+Char(13)
End for
    ` ...
```

See Also

ASCII Codes, ISO to Mac, SEND HTML FILE, SEND PACKET, USE CHARACTER SET.

Mac to Win (text) → String

Parameter	Type		Description
text	String	→	Text expressed using Mac OS ASCII map
Function result	String	←	Text expressed using Windows ANSI map

Compatibility Note: This command only works when the database is executed in ASCII compatibility mode. In Unicode mode, it does nothing (the text string is returned without modification). Beginning with version 11 of 4D, this command is thus 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.

Description

The Mac to Win command returns a text expressed using the Windows ANSI map that is equivalent to the text you pass in text, which is expressed using the Mac OS ASCII map.

This command expects a Text type parameter that is expressed using the Mac OS ASCII map.

Generally, when running on Windows, you do not need to use this command to convert ASCII codes. In ASCII compatibility mode (non-Unicode), when you copy or paste text between 4D and Windows or when you import or export data, 4D automatically performs these conversions. However, when you use disk read/write commands such as SEND PACKET or RECEIVE PACKET, you need to explicitly perform ASCII conversions. This is the main purpose of the Mac to Win command.

Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are Mac OS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

Note: This command replaces the CR (Carriage return) characters with CRLF (Carriage return + Line feed, ASCII codes 13 and 10) characters. Consequently, the text returned may be longer than the original text.

Example

On Windows, when you write characters into a document using SEND PACKET, if you do not use an output ASCII map for filtering characters from Mac OS to Windows (see USE CHARACTER SET), you need to convert the text from Mac OS to Windows yourself. You can do it this way:

```
` ...
SEND PACKET ($vhDocRef;Mac to Win(vtSomeText))
` ...
```

See Also

ASCII Codes, SEND PACKET, USE CHARACTER SET, Win to Mac.

Match regex

version 11

Match regex (pattern; aString{; start; pos_found; length_found; *}) → Boolean

Parameter	Type		Description
pattern	Text	→	Regular expression
aString	Text	→	String in which search will be done
start	Number	→	Position in aString where search will start
pos_found	Longint Var | Longint Array	←	Position of occurrence
length_found	Longint Var | Longint Array	←	Length of occurrence
*	*	→	If passed: only searches at position indicated
Function result	Boolean	←	True = search has found an occurrence; Otherwise, False.

Description

The Match regex command can be used to check 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 in which 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

Examples

This command can be used in several ways, as illustrated by the following examples:

1. Search for complete equality:
vfound:=Match regex(pattern;mytext)

```
QUERY BY FORMULA([Employees];Match regex(".*smith.*"; [Employees]name))
```

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))
```

3. Search with support of “capture groups”:
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
```

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 Handling

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

String

version 11 (Modified)

Num (expression; separator) → Number

Parameter	Type		Description
expression	String | Boolean | Num	→	String for which to return the numeric form, or Boolean to return 0 or 1, or Numeric expression
separator	String	→	Decimal separator
Function result	Number	←	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 can be used to designate 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, Num ignores the alphabetic characters. Thus, Num transforms the string "a1b2c3" into the number 123.

Note: Only the first 32 characters of expression are evaluated.

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 on "e", conversion might give different results under Mac OS and under Windows.

The separator parameter can be used to designate 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.

Examples

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
```

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))
```

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
```

See Also

GET SYSTEM FORMAT, Logical Operators, String, String Operators.

Position (find; aString{; start{; lengthFound{; *}}}) → Number

Parameter	Type		Description
find	String	→	String to find
aString	String	→	String in which to search
start	Number	→	Position in string where search will start
lengthFound	Longint	←	Length of string found
*	*	→	If passed: diacritic-sensitive search
Function result	Number	←	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 (diacritic-sensitive mode, see below), these letters are not considered as equivalent (æ # ae); in diacritic-sensitive mode, lengthFound is always equal to the length of find (if an occurrence is found).

The * parameter, if passed, indicates that the search must be diacritic-sensitive, i.e. it must take the case of the characters and diacritic marks into account (a # A, à # a, etc.)

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.

Examples

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
```

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)
```

See Also

Comparison Operators, Substring.

Replace string (source; oldString; newString{; howMany{; *}}) → String

Parameter	Type		Description
source	String	→	Original string
oldString	String	→	Characters to replace
newString	String	→	Replacement string (if empty string, occurrences are deleted)
howMany	Number	→	How many times to replace If omitted, all occurrences are replaced
*	*	→	If passed: diacritical evaluation
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 is not case sensitive and does not take accented characters into account (a=A, a=à, etc.). If you pass the asterisk * as the last parameter, you indicate that the evaluation of the characters must be diacritical, in other words, it must be case sensitive and take accented characters into account (a#A, a#à...).

Examples

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(9);",")  `Replaces tabs in vtOtherVar with commas
```

2. The following example eliminates CRs and TABs from the text in vtResult:

```
vtResult:=Replace string(Replace string(vtResult;Char(13);"");Char(9);"")
```

3. The following example illustrates the use of the * parameter:

```
vtResult:=Replace string("Crème brûlée";"Brulee";"caramel") `Result gets "Crème caramel"
vtResult:=Replace string("Crème brûlée";"Brulee";"caramel";*) `Result gets "Crème brûlée"
```

See Also

Change string, Delete string, Insert string.

String (expression{; format}) → String

Parameter	Type		Description
expression		→	Expression for which to return the string form (can be Real, Integer, Long Integer, Date, Time String, Text or Boolean)
format	String | Number	→	Display format
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.

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
String(2^15) ` Use default format	32768 (Default format used here)
String(2^15;"###,##0 Inhabitants")	32,768 Inhabitants
String(1/3;"##0.00000")	0.33333
String(1/3) ` Use default format used here)	0.3333333333333333 (Default format
String(Arctan(1)*4) used here)	3.1415926535897931 (Default format
String(Arctan(1)*4;"##0.00")	3.14
String(-1;"&x")	0xFFFFFFFF
String(-1;"&$")	$FFFFFFFF
String(0 ?+ 7;"&x")	0x80
String(0 ?+ 7;"&$")	$80
String(0 ?+ 14;"&x")	0x4000

String(0 ?+ 14;"&$")	$4000
String(Num(1=1);"True;;False")	True
String(Num(1=2);"True;;False")	False

The format is specified in the same way as it would be for a number field on a form. See 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.

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 following constants ("Date Display Formats" theme):

Number	Constant	Example
1	System date short	12/29/2006
2	System date abbreviated	Sun, Dec 29, 2006
3	System date long	Sunday, December 29, 2006
4	Internal date short special	12/29/06 or 12/29/1896 or 12/29/2096
5	Internal date long	December 29, 2006
6	Internal date abbreviated	Dec 29, 2006
7	Internal date short	12/29/2006
8	ISO Date Time	2006-12-29T00:00:00
100	Blank if null	"" instead of 0

Notes:
•The ISO Date Time format corresponds to the ISO8601 standard. This format contains a date and a time. For example, the date May 31, 2006 at 1:20 p.m. is written 2006-05-31T13:20:00. It is used for XML processing and with Web services. 4D does not permit storing a date and a time in a single field. However, it is possible to handle dates in this format using the String command.
• The Blank if null 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 date is 12/29/2006:

```
$vsResult:=String(Current date)  ` $vsResult gets "12/29/06"
$vsResult:=String(Current date;Month Date Year)  ` $vsResult gets "December 29, 2006"
$vsResult:=String(Current date;ISO Date Time)  ` $vsResult gets "2006-12-29T00:00:00"
```

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 ("Time Display Formats" theme):

Number	Constant	Example
1	HH:MM:SS	01:02:03
2	HH:MM	01:02
3	Hour Min Sec	1 hour 2 minutes 3 seconds
4	Hour Min	1 hour 2 minutes
5	HH MM AM PM	1:02 AM
6	MM SS	62:03
7	Min Sec	62 minutes 3 seconds
8	ISO Date Time	0000-00-00T01:02:03
9	System time short	01:02:03
10	System time long abbreviated	1•02•03 AM (*Mac only*)
11	System time long	1:02:03 AM HNEC (*Mac only*)
100	Blank if null	"" instead of 0

Notes:
• For more information about the ISO Date Time format, refer to the note above.
• The Blank if null 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 gets "17 hours 30 minutes 45 seconds"
$vsResult:=String(Current time;Hour Min Sec)
```

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.

See Also

Date, Num, Time string.

Substring

String

version 3

Substring (source; firstChar{; numChars}) → String

Parameter	Type		Description
source	String	→	String from which to get substring
firstChar	Number	→	Position of first character
numChars	Number	→	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 ("").

Examples

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
```

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="")
```

See Also

Position.

Uppercase (aString; *) → String

Parameter	Type		Description
aString	String	→	String to convert to uppercase
*	*	→	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.

Examples

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"
```

2. See the example for Lowercase.

See Also

Lowercase.

Win to Mac (text) → String

Parameter	Type		Description
text	String	→	Text expressed using Windows ANSI map
Function result	String	←	Text expressed using Macintosh ASCII map

Compatibility Note: This command only works when the database is executed in ASCII compatibility mode. In Unicode mode, it does nothing (the text string is returned without modification). Beginning with version 11 of 4D, this command is thus 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.

Description

The Win to Mac command returns text expressed using the Mac OS ASCII map that is equivalent to the text you pass in Text, which is expressed using the Windows ANSI map.

This command expects a text parameter that is expressed using the Windows ANSI map.

Generally, when running on Windows, you do not need to use this command to convert ASCII codes. In ASCII compatibility mode (non-Unicode), when you copy or paste text between 4D and Windows or when you import or export data, 4D automatically performs these conversions. However, when you use disk read/write commands such as SEND PACKET or RECEIVE PACKET, you need to explicitly perform ASCII conversions. This is the main purpose of the Win to Mac command.

Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are Mac OS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

Note: This command replaces the CRLF (Carriage return + Line feed, ASCII codes 13 and 10) characters with the CR (Carriage return) characters. Consequently, the text returned may be shorter than the original text.

Example

When you read characters from a Windows document using RECEIVE PACKET, if you do not use an input ASCII map for filtering characters from Windows to Mac OS (see USE CHARACTER SET), you need to convert the text from Windows to Mac OS yourself. You can do it this way:

```
` ...
RECEIVE PACKET ($vhDocRef;vtSomeText;16*1024)
vtSomeText:=Win to Mac(vtSomeText)
` ...
```

See Also

ASCII Codes, Mac to Win, RECEIVE PACKET, USE CHARACTER SET.

50

Structure Access

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.

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.

Counting tables and fields
Since version 11 of 4D, it is possible to delete tables and fields. This possibility means that the algorithms used in previous versions for counting tables and fields need to be modified. It is now necessary to use algorithms combining the Get last table number and Get last field number and 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
```

See Also

Field, GET FIELD PROPERTIES, Get last field number, Get last table number, Pointers, SET INDEX, Table, Table name.

CREATE INDEX (theTable; fieldsArray; indexType; indexName{; *})

Parameter	Type		Description
aTable	Table	→	Table for which to create an index
fieldsArray	Array pointer	→	Pointer(s) to field(s) to be indexed
indexType	Integer	→	Type of index to create: -1 = Keywords, 0 = default, 1 = Standard B-Tree, 3 = Cluster B-Tree
indexName	Text	→	Name of index to create
*	*	→	If passed = asynchronous indexing

Description

The CREATE INDEX command can be used to create:
• 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 is used to define the type of index to be created. You can pass one of the following constants, found in the "Index Type" theme:
• Keywords Index (-1): Keyword type index. Remember that keywords cannot be composite: you must pass only one field in the fieldsArray array.
• Default Index Type (0): In this case, 4D sets the index type (except for keyword indexes) that is optimal according to the contents of the field.
• Standard BTree Index (1): Index of the standard B-Tree type. This multipurpose index type is used in previous versions of 4D.
• Cluster BTree Index (3): Index of the B-Tree type using clusters. This index type is optimal when the index contains few keys, i.e. when the same values occur frequently in the data.

In the indexName parameter, pass the name of the index to be created. This name is mandatory; if you pass an empty string, an error is generated. If the indexName index already exists, the command does nothing.

The optional * parameter, when it is passed, is used to carry out 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 errorhandlingmethod.

Examples

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")
```

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")
```

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")
```

See Also

DELETE INDEX, SET INDEX.

DELETE INDEX (fieldPtr | indexName{; *})

Parameter	Type		Description
fieldPtr | indexName	Pointer | Text	→	Pointer to field whose indexes are to be deleted or Name of index to be deleted
*	*	→	If passed = asynchronous operation

Description

The DELETE INDEX command is used to delete 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. If the field is included in one or more composite indexes, they will also be deleted.
• If you pass the name of an index (indexName), only the designated index will be deleted. This can consist of keyword indexes or standard indexes. If the field contains other indexes or belongs to other composite indexes, they will not be deleted.

The optional * parameter, when it is passed, is used to carry out the 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")
```

See Also

CREATE INDEX, SET INDEX.

Field (tableNum | fieldPtr{; fieldNum}) → Number | Pointer

Parameter	Type		Description
tableNum | fieldPtr	Number | Pointer	→	Table number or Field pointer
fieldNum	Number	→	Field number, if Table number is passed
Function result	Number | Pointer	←	Field number, if Field pointer is passed Field pointer, if Table and Field numbers are passed

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.

Examples

1. The following example sets the fieldPtr variable to a pointer to the second field in the third table:

```
FieldPtr:=Field(3; 2)
```

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)
```

3. The following example sets the FieldNum variable to the field number of [Table3]Field2:

```
FieldNum:=Field(->[Table3]Field2)
```

See Also

Field name, GET FIELD PROPERTIES, Get last field number, Table.

Field name (fieldPtr | tableNum{; fieldNum}) → String

Parameter	Type		Description
fieldPtr | tableNum	Number	→	Field pointer or Table number
fieldNum	Number	→	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.

Examples

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)
```

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)
```

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.")
```

See Also

Field, Get last field number, Table name.

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 Subtable and 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 subtable and 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 subtable and BLOB.

See Also

GET FIELD PROPERTIES, GET RELATION PROPERTIES, GET TABLE PROPERTIES.

GET FIELD PROPERTIES (fieldPtr | tableNum{; fieldNum}; fieldType{; fieldLength{; indexed{; unique{; invisible}}}})

Parameter	Type		Description
fieldPtr | tableNum	Pointer | Number	→	Table number or Field pointer
fieldNum	Number	→	Field number if Table number is passed
fieldType	Number	←	Type of field
fieldLength	Number	←	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:

Constant	Type	Value
Is Alpha Field	Long Integer	0
Is Text	Long Integer	2
Is Real	Long Integer	1
Is Integer	Long Integer	8
Is LongInt	Long Integer	9
Is Date	Long Integer	4
Is Time	Long Integer	11
Is Boolean	Long Integer	6
Is Picture	Long Integer	3
Is Subtable	Long Integer	7
Is BLOB	Long Integer	30

• 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 Unique attribute can be set only to indexed fields.

• 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...).

Examples

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)
```

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)
```

See Also

Field, Field name, SET INDEX.

Get last field number

version 11 (Modified)

Get last field number (tableNum | tablePtr) → Number

Parameter	Type		Description
tableNum | tablePtr	Number | Pointer	→	Table number or Pointer to table
Function result	Number	←	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 ($vlField;1;Size of array(asFields);1;-1)
    If(Is field number valid($vlTable;$vlField))
        asFields{$vlField}:=Field name($vlTable;$vlField)
    Else
        DELETE FROM ARRAY(asFields;$vlField)
    End if
End for
```

See Also

Field name, GET FIELD PROPERTIES, Get last table number, Is field number valid.

Get last table number → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Highest table number in the database

Description

Get last table number returns the number of 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)
For ($vlTable; 1; Size of array(asTables);1;-1)
        asTables {$vlTable}:=Table name ($vlTable)
    Else
        DELETE FROM ARRAY(asTables; $vlTable)
    End if
End for
```

See Also

Get last field number, Is table number valid, Table name.

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.

See Also

GET FIELD ENTRY PROPERTIES, GET FIELD PROPERTIES, GET TABLE PROPERTIES, SET AUTOMATIC RELATIONS, SET FIELD RELATION.

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	←	True = Trigger "On loading a record" activated; otherwise, False

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.
• The trigLoadRec parameter returns True if the "On loading a record" trigger has been set for this table, else False.

See Also

GET FIELD ENTRY PROPERTIES, GET FIELD PROPERTIES, GET RELATION PROPERTIES.

Is field number valid

Is field number valid (tableNum | tablePtr; fieldNum) → Boolean

Parameter	Type		Description
tableNum | tablePtr	Num | 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.

See Also

Get last table number, Is table number valid.

Is table number valid

version 11

Is table number valid (tableNum) → Boolean

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.

See Also

Get last table number, Is field number valid.

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

The SET INDEX accepts two syntaxes:
• If you pass a Boolean in theindex 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 non-composite indexes 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:
• Keywords Index (-1): Index of keywords which permits word-by-word indexing of field contents. This type of index can only be used with fields of the Text or Alpha type.
• Default Index Type (0): In this case, 4D defines the index type (excluding keywords indexes) that is the most optimized according to the contents of the field.
• Standard BTree Index (1): Standard B-Tree type index. This multi-purpose index type is used in previous versions of 4D.
• Cluster BTree Index (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.

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.

Examples

1. The following example indexes the field [Customers]ID:

```
UNLOAD RECORD([Customers])
SET INDEX ([Customers]ID; True)
```

2. You want to index the [Customers]Name field with the fast mode:

```
SET INDEX ([Customers]Name; True;100)
```

3. Creation of a keywords index:

```
SET INDEX([Books]Summary;Keywords Index)
```

See Also

CREATE INDEX, DELETE INDEX, GET FIELD PROPERTIES, ORDER BY, QUERY.

Table (tableNum | aPtr) → Pointer | Number

Parameter	Type		Description
tableNum | aPtr	Number | Pointer	→	Table number, or Table pointer, or Field pointer
Function result	Pointer | Number	←	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.

Examples

1. This example sets the tablePtr variable to a pointer to the third table of the database:

```
TablePtr:=Table(3)
```

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)
```

3. This example sets the tableNum variable to the table number of [Table3]:

```
TableNum:=Table(->[Table3])
```

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)
```

See Also

Field, Get last table number, Pointers, Table name.

Table name (tableNum | tablePtr) → String

Parameter	Type		Description
tableNum | tablePtr	Number | 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 of 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
```

See Also

Field name, Get last table number, Table.

51

Subrecords

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.

Description

ALL SUBRECORDS makes all the subrecords of subtable the current subselection. If a current parent record does not exist, ALL SUBRECORDS has no effect. When a parent record is first loaded, the subselection contains all subrecords. A subselection may not contain all subrecords after ADD SUBRECORD, QUERY SUBRECORDS, or DELETE SUBRECORD is executed.

Example

The following example selects all subrecords to ensure that they are all included in the sum:

```
ALL SUBRECORDS ([Stats]Sales)
TotalSales := Sum ([Stats]Sales'Dollars)
```

See Also

QUERY SUBRECORDS, Records in subselection.

APPLY TO SUBSELECTION

version 11 (Modified)

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.

Description

APPLY TO SUBSELECTION applies statement to each subrecord in the current subselection of subtable. The statement may be a statement or a method. If the statement modifies a subrecord, the modified subrecord is written to disk only when the parent record is written. If the subselection is empty, APPLY TO SUBSELECTION has no effect.

APPLY TO SUBSELECTION can be used to gather information from the subselection or to modify the subselection.

Example

The following example capitalizes the first names in [People]Children:

```
ALL SUBRECORDS ([People]Children)
APPLY TO SUBSELECTION([People]Children;[People]Children'Name:=Uppercase
                (Substring([People]Children'Name;1;1))+Lowercase(Substring([People]
                                                Children'Name;2)))
```

Note: The statement has been put on several lines for clarity in documentation only.

See Also

ALL SUBRECORDS, EDIT FORMULA, QUERY SUBRECORDS, SAVE RECORD.

Before subselection

version 11 (Modified)

Before subselection (subtable) → Boolean

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.

Description

Before subselection returns True when the current subrecord pointer is before the first subrecord of subtable. Before subselection is used to check whether or not PREVIOUS SUBRECORD has moved the pointer before the first subrecord. If the current subselection is empty, Before subselection returns True.

Example

The following example is an object method for a button. When the button is clicked, the pointer moves to the previous subrecord. If the pointer is before the first subrecord, it moves to the last subrecord:

```
PREVIOUS SUBRECORD ([People]Children)  ` Move to the previous subrecord
If (Before subselection ([People]Children)  ` If we have gone too far...
   LAST SUBRECORD ([People]Children)  ` move to the last subrecord
End if
```

See Also

PREVIOUS SUBRECORD.

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.

Description

CREATE SUBRECORD creates a new subrecord for subtable and makes the new subrecord the current subrecord. The new subrecord is saved only when the parent record is saved. The parent record can be saved by a command such as SAVE RECORD or by the user accepting the record. If there is no current record, CREATE SUBRECORD has no effect. To add a new subrecord through a subrecord input form, use ADD SUBRECORD.

Example

The following example is an object method for a button. When it is executed (that is, when the button is clicked), it creates new subrecords for children in the [People] table. The Repeat loop lets the user add children until the Cancel button is clicked. The form displays the children in an subform, but will not allow direct data entry into the subtable because the Enterable option has been turned off:

```
Repeat
      ` Get the child's name
   vChild := Request("Name (cancel when done):")
      ` If the user clicked OK
   If (OK = 1)
         ` Add a new subrecord for a child
      CREATE SUBRECORD([People]Children)
         ` Assign child's name to the subfield
      [People]Children'Name:=vChild
   End if
Until (OK=0)
```

See Also

ADD SUBRECORD, DELETE SUBRECORD, MODIFY SUBRECORD, SAVE RECORD.

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.

Description

DELETE SUBRECORD deletes the current subrecord of subtable. If there is no current subrecord, DELETE SUBRECORD has no effect. After the subrecord is deleted, the current subselection for subtable is empty. As a result, DELETE SUBRECORD can't be used to scan through a subselection and delete selected subrecords.

The deletion of subrecords is not permanent until the parent record is saved. Deleting a parent record automatically deletes all its subrecords.

To delete a subselection, create the subselection you want to delete, delete the first subrecord, create the subselection again, delete the first subrecord, and so on.

Examples

1. The following example deletes all the subrecords of a subtable:

```
ALL SUBRECORDS([People]Children)
While (Records in subselection([People]Children)>0)
    DELETE SUBRECORD([People]Children)
    ALL SUBRECORDS([People]Children)
End while
```

2. The following example deletes the subrecords in which the age of the child is greater than or equal to 12, from the [People]Children subtable :

```
ALL RECORDS([People])  ` Select all the records
For ($vlRecord;1;Records in selection([People]))  ` For all the records in the table
      ` Query all records that have subrecords with the criteria
   QUERY SUBRECORDS([People]Children;[People]Children'Age>=12)
      ` Loop until no subrecords are left by the query
   While (Records in subselection([People]Children)>0)
         ` Delete the subrecord
      DELETE SUBRECORD([People]Children)
         ` Query again
      QUERY SUBRECORDS([People]Children;[People]Children'Age>=12)
   End while
   SAVE RECORD([People])  ` Save the parent record
   NEXT RECORD([People])
End for
```

See Also

ALL SUBRECORDS, QUERY SUBRECORDS, Records in subselection, SAVE RECORD.

End subselection

version 11 (Modified)

End subselection (subtable) → Boolean

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.

Description

End subselection returns True when the current subrecord pointer is after the end of the current subselection of subtable. End subselection is used to check whether or not NEXT SUBRECORD has moved the pointer after the last subrecord. If the current subselection is empty, End subselection returns True.

Example

The following example is an object method for a button. When the button is clicked, the pointer moves to the next subrecord. If the pointer is after the last subrecord, it moves to the first subrecord:

```
NEXT SUBRECORD ([People]Children)  ` Move to the next subrecord
If (End subselection ([People]Children))  ` If we have gone too far...
   FIRST SUBRECORD ([People]Children)  ` move to the first subrecord
End if
```

See Also

NEXT SUBRECORD.

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.

Description

FIRST SUBRECORD makes the first subrecord of the current subselection of subtable the current subrecord. All query, selection, and order by commands also set the current subrecord to the first subrecord. If the current subselection is empty, FIRST SUBRECORD has no effect.

Example

The following example concatenates the first and last names in child records stored in a subtable. It copies the names into the array atNames:

```
    ` Create an array to hold the names
ARRAY TEXT (atNames; Records in subselection ([People]Children))
    ` Start at the first subrecord and loop once for each child
FIRST SUBRECORD ([People]Children)
For ($vlSub; 1; Records in subselection ([People]Children))
    atNames{$vlSub} := [People]Children'First Name+" "+ [People]Children'Last Name
    NEXT SUBRECORD ([People]Children)
End for
```

See Also

LAST SUBRECORD, NEXT SUBRECORD, PREVIOUS SUBRECORD.

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.

Description

LAST SUBRECORD makes the last subrecord of the current subselection of subtable the current subrecord. If the current subselection is empty, LAST SUBRECORD has no effect.

Example

The following example concatenates the first and last names in child records stored in a subtable. It copies the names into an array, called atNames. It is the same as the example for FIRST SUBRECORD except that it moves through the subrecords from last to first:

```
    ` Create an array to hold the names
ARRAY TEXT (atNames; Records in subselection ([People]Children))
    ` Start at the last subrecord and loop once for each child
LAST SUBRECORD ([People]Children)
For ($vlSub;1;Records in subselection ([People]Children))
    atNames{$vlSub}:=[People]Children First Names + " " + [People]Children Last Names
    PREVIOUS SUBRECORD ([People]Children)
End for
```

See Also

FIRST SUBRECORD, NEXT SUBRECORD, PREVIOUS SUBRECORD.

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.

Description

NEXT SUBRECORD moves the current subrecord pointer to the next subrecord in the current subselection of subtable. If NEXT SUBRECORD moves the pointer past the last subrecord, End subselection returns TRUE, and there is no current subrecord. If End subselection returns TRUE, use FIRST SUBRECORD or LAST SUBRECORD to move the pointer back into the current subselection. If the current subselection is empty, or Before subselection returns TRUE, NEXT SUBRECORD has no effect.

Example

See the example for FIRST SUBRECORD.

See Also

FIRST SUBRECORD, LAST SUBRECORD, PREVIOUS SUBRECORD.

ORDER SUBRECORDS BY

version 11 (Modified)

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 <		→	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.

Description

ORDER SUBRECORDS BY sorts the current subselection of subtable. It sorts only the subselection of the subtable contained in the current parent record.

The direction parameter specifies whether to sort subfield in ascending or descending order. If direction is the “greater than” symbol (>), the subrecords are ordered in ascending order. If direction is the “less than” symbol (<), the subrecords are ordered in descending order.

You can specify more than one level of sort by including more subfields and sort symbols. After the sort is completed, the first subrecord of the sorted subselection is the current subrecord. Sorting subrecords is a dynamic process. Subrecords are never saved in their sorted order. If neither a current record nor a higher-level subrecord exists, ORDER SUBRECORDS BY has no effect.
If a form contains a subform that is to be printed in a fixed frame, this command needs to be called just once before printing in the Before phase of the parent form method.

Example

The following example sorts the [Stats]Sales subtable into ascending order, based on the SalesDollars subfield:

```
ORDER SUBRECORDS BY ([Stats]Sales; [Stats]Sales'Dollars; >)
```

See Also

QUERY SUBRECORDS.

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.

Description

PREVIOUS SUBRECORD moves the current subrecord pointer to the previous subrecord in the current subselection of subtable. If PREVIOUS SUBRECORD moves the pointer before the first subrecord, Before subselection returns TRUE, and there is no current subrecord. If Before subselection returns TRUE, use FIRST SUBRECORD or LAST SUBRECORD to move the pointer back into the current subselection. If the current subselection is empty, or End subselection returns TRUE, PREVIOUS SUBRECORD has no effect.

Example

See the example for LAST SUBRECORD.

See Also

FIRST SUBRECORD, LAST SUBRECORD, NEXT SUBRECORD.

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.

Description

QUERY SUBRECORDS queries subtable and creates a new subselection. This is the only command that queries subrecords and returns a selection of subrecords. The queryFormula is applied to each subrecord in subtable. If the formula evaluates as TRUE, the subrecord is added to the new subselection. When the query is complete, QUERY SUBRECORDS makes the first subrecord the current subrecord of subtable.

Remember that QUERY SUBRECORDS queries only the subrecords of the subtable contained in the currently selected parent record, and not all the subrecords associated with the records of the parent table. QUERY SUBRECORDS does not change the current parent record.

Typically, queryFormula tests a subfield against a variable or a constant, using a relational operator. The queryFormula can contain multiple tests that are joined by AND conjunctions (&) or OR conjunctions (|). Also, the queryFormula can be a function or contain a function. The wildcard character (@) can be used with string arguments.

If neither a current record nor a higher-level subrecord exists, QUERY SUBRECORDS has no effect.

Example

The following example queries for children older than 10 years:

```
QUERY SUBRECORDS ([People]Children; [People]Children'Age>10)
```

See Also

ALL SUBRECORDS, ORDER SUBRECORDS BY, Records in subselection.

Records in subselection (subtable) → Number

Parameter	Type		Description
subtable	Subtable	→	Subtable for which to count number of subrecords
Function result	Number	←	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.

Description

Records in subselection returns the number of subrecords in the current subselection of subtable. Records in subselection applies only to subrecords in the current record. It is the subrecord equivalent of Records in selection. The result is undefined if no parent record exists.

Example

The following example selects all the subrecords and displays the number of children for the parent record:

```
   ` Select all children, then display how many
ALL SUBRECORDS ([People]Children)
ALERT ("Number of children: "+String(Records in subselection ([People]Children)))
```

See Also

ALL SUBRECORDS, QUERY SUBRECORDS.

52

System Documents

System Documents

version 11 (Modified)

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 NetBEUI (Windows) or AFP (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 A: and B: are used to designate the 5 1/4 or 3 1/2 floppy drives. Usually C: designates the volume you use for booting your system (unless you configure your PC otherwise). Then the letters D: 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 **directory (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**.

Document Type and Creator

On Windows, a document has a **type**. On Macintosh, a document also has a **type** and may have in addition a **creator**. The type of a document generally indicates what the document is or what it contains. For instance, a text document contains some text (without style variations).

The type of a document is determined by the suffix (called the **file extension**) attached to the document name. For instance, .TXT or .TEXT is the file extension for text documents. This principle is the same under Mac OS X, however for reasons of compatibility with previous versions of the system, the type of a document is determined by the **file type** property if it has been specified. This property is a 4-character signature (not displayed at the Finder level). For instance, the file type of a text document is "TEXT".

In addition,a document may have a **creator**, which designates the application that created the document. This concept does not exist on Windows. The creator of a document is determined by the **file creator** property. If a document has both the type and creator properties, Mac OS will take them into account regardless of the document extension.

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.

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.

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)

• 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)
```

• Using the right directory separator symbol

On Windows, a directory level is symbolized by an backslash (\). On Macintosh, a folder level is symbolized by a colon (:). Depending on which platform you are running, the Directory symbol project method listed here returns the code of the correct directory symbol (character).

```
` Directory symbol Project Method
` Directory symbol -> Integer
` Directory symbol -> Code of "\" (Windows) or ":" (Mac OS)

C_INTEGER($0)

If (On Windows )
   $0:=Character code("\\")
Else
   $0:=Character code(":")
End if
```

• 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_STRING(255;$1;$0)
C_INTEGER($viLen;$viPos;$viChar;$viDirSymbol)

$viDirSymbol:=Directory symbol
$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_STRING(255;$1;$0)
C_STRING(1;$vsDirSymbol)
C_INTEGER($viLen;$viPos;$viChar;$viDirSymbol)

$viDirSymbol:=Directory symbol
$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
```

See Also

Append document, CLOSE DOCUMENT, COPY DOCUMENT, Create document, CREATE FOLDER, DELETE DOCUMENT, Document creator, DOCUMENT LIST, Document type, FOLDER LIST, Get document position, GET DOCUMENT PROPERTIES, Get document size, MAP FILE TYPES, MOVE DOCUMENT, Open document, Select document, SET DOCUMENT CREATOR, SET DOCUMENT POSITION, SET DOCUMENT PROPERTIES, SET DOCUMENT SIZE, SET DOCUMENT TYPE, Test path name, VOLUME ATTRIBUTES, VOLUME LIST.

Append document (document{; fileType}) → DocRef

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 allows you to open 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
```

See Also

Create document, Open 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
```

See Also

Append document, Create document, Open document.

COPY DOCUMENT (sourceName; destinationName{; *})

Parameter	Type		Description
sourceName	String	→	Name of document to be copied
destinationName	String	→	Name of copied document
*		→	Override existing document if any

Description

The COPY DOCUMENT command copies the document specified by sourceName to the location specified by destinationName.

Both sourceName and destinationName can be a name referring to a document located in the database folder or a pathname referring to a document in relation to the root level of a volume.

An error will occur if there is already a document named destinationName unless you specify the optional * parameter instructing COPY DOCUMENT to delete and override the destination document.

Examples

1. The following example duplicates a document in its own folder:

```
COPY DOCUMENT("C:\\FOLDER\\DocName";"C:\\FOLDER\\DocName2")
```

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")
```

3. The following example copies a document from one volume to another one:

```
COPY DOCUMENT("C:\\FOLDER\\DocName";"F:\\Archives\\DocName.OLD")
```

4. The following example duplicates a document in its own folder overriding an already existing copy:

```
COPY DOCUMENT("C:\\FOLDER\\DocName";"C:\\FOLDER\\DocName2";*)
```

See Also

MOVE DOCUMENT.

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 or Sets

The OK system variable is set to 1 if the command execution was successful; otherwise it is set to 0.

See Also

RESOLVE ALIAS.

Create document (document{; fileType}) → DocRef

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 is displayed 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: http://developer.apple.com/documentation/Carbon/Conceptual/understanding_utis/index.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 SEND PACKET and RECEIVE 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.

Examples

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
```

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 or 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.

See Also

Append document, Open document.

CREATE FOLDER (folderPath)

Parameter	Type		Description
folderPath	String	→	Pathname to new folder to create

Description

The CREATE FOLDER command creates a folder according to the pathname you pass in folderPath.

If you pass a name, the folder is created in the folder of the database. If you pass a pathname, it must refer to a valid path up to the name of the folder you want to create, starting at the root level of a volume or at the level of the database folder.

Examples

1. The following example creates the “Archives” folder in the folder of the database:

```
CREATE FOLDER("Archives")
```

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")
```

3. The following example creates the “Archives” folder at the root level of the C volume:

```
CREATE FOLDER("C:\\Archives")
```

4. The following example will fail if there is no “NewStuff” folder at the root level of the C volume:

```
   ` WRONG, cannot create two folder levels in one call
CREATE FOLDER("C:\\NewStuff\\Pictures")
```

See Also

FOLDER LIST, Test path name.

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.

Examples

1. The following example deletes the document named Note:

```
DELETE DOCUMENT ("Note") ` Delete the document
```

2. See example for the command APPEND DATA TO PASTEBOARD.

System Variables or 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 (folder)

Parameter	Type		Description
folder	String	→	Name or full path of the folder to be deleted

Description

The DELETE FOLDER command deletes the folder whose name or full path has been passed in folder.

Only empty folders can be deleted by this command.
• If you try to delete a folder containing files, the -47 error is generated (Attempt to delete a non-empty folder).
• If you pass in folder a file path or an empty string or the path to a non-existing folder, the command does nothing and generates a -43 error (File not found).

You can detect these errors through a method installed by the ON ERR CALL command.

See Also

DELETE DOCUMENT.

Document creator (document) → String

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.

See Also

Document type, SET DOCUMENT CREATOR.

DOCUMENT LIST (pathname; documents)

Parameter	Type		Description
pathname	String	→	Pathname to volume, directory or folder
documents	String array	←	Names of the documents present at this location

Description

The DOCUMENT LIST command populates the Text or String array documents with the names of the documents located at the pathname you pass in pathname.

Note: The pathname parameter only accepts absolute pathnames.

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.

See Also

FOLDER LIST, VOLUME LIST.

Document type (document) → String

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 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 the 4-characters file type of the document (i.e. 'TEXT' for a Text document, 'APPL' for a double-clickable application and so on).

See Also

Document creator, GET DOCUMENT PROPERTIES, MAP FILE TYPES, SET DOCUMENT TYPE.

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.

See Also

DOCUMENT LIST, VOLUME LIST.

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
		←	Document icon
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 filePath. filePath can specify a file of any type (executable, document, shortcut or alias, etc.) or a folder.

filePath should contain 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 filePath, the standard Open File dialog box is presented. 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 in icon a 4D picture field or variable. After the command is executed, this parameter contains the icon of the file (PICT format).

The optional size parameter allows you to set 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 (docRef) → Number

Parameter	Type		Description
docRef	DocRef	→	Document reference number
Function result	Number	←	File position (expressed in bytes) from the beginning of the file

Description

This command operates only on a document 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.

See Also

RECEIVE PACKET, SEND PACKET, SET DOCUMENT POSITION.

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:

Documents	
Number	**L**
Subject	**A**
Theme	**A**
Description	**T**
Creation Stamp	**L**
Modification Stamp	**L**

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
    : (Database event=Save New Record Event)
        [Documents]Creation Stamp:=Time stamp
        [Documents]Modification Stamp:=Time stamp
    : (Database 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($vsPath) # Is a directory)
   CREATE FOLDER($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)))
```

See Also

Document creator, Document type, SET DOCUMENT PROPERTIES.

Get document size (document{; *}) → Number

Parameter	Type		Description
document	DocRef | String	→	Document reference number or Document name
*		→	On Mac OS only: - if omitted, size of data fork - if specified, size of resource fork
Function result	Number	←	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.

See Also

Get document position, SET DOCUMENT POSITION, SET DOCUMENT SIZE.

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 Append document, Create document, Create resource file, Open resource file 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
```

See Also

Append document, Create document, Create resource file, Open resource file.

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.

Examples

1. The following example renames the document DocName:

```
MOVE DOCUMENT("C:\\FOLDER\\DocName";"C:\\FOLDER\\NewDocName")
```

2. The following example moves and renames the document DocName:

```
MOVE DOCUMENT("C:\\FOLDER1\\DocName";"C:\\FOLDER2\\NewDocName")
```

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.

See Also

COPY DOCUMENT.

Open document (document{; fileType{; mode}}) → DocRef

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	Integer	→	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 and the mode parameter is omitted, Open document opens the document in Read mode and sets the OK variable to 1.
- If the document is already open and you try to open it in Write mode, an error is generated.
- 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:
http://developer.apple.com/documentation/Carbon/Conceptual/understanding_utis/index.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
Read and Write (default value)	Integer	0
Write Mode	Integer	1
Read Mode	Integer	2
Get Pathname	Integer	3

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.

Examples

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
```

2. You can now 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 Variable 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 the file defined in document is not found or 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.

See Also

Append document, Create document.

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.

See Also

CREATE ALIAS.

System Variables or 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 (directory; fileTypes; title; options{; selected}) → String

Parameter	Type		Description
directory	Text | Longint	→	• Directory access path to display by default in the document selection dialog box, or • Empty string to display default user folder ("My documents" under Windows, "Documents" under Mac OS), or • 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:
http://developer.apple.com/documentation/Carbon/Conceptual/understanding_utis/index.html.

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
Multiple files	Longint	1
Package open	Longint	2
Package selection	Longint	4
Alias selection	Longint	8
Use Sheet Window	Longint	16

You can pass one or a combination of constants.
• Multiple files: 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 (Mac OS only): Authorizes the opening of packages and the viewing of their contents. By default, if this constant is not used, the command will not allow the opening of packages.
• Package selection (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. In this case, it is impossible to open or select a software package (even if the Package open constant is passed.
• Alias selection: 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.
• Use Sheet Window (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.

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

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)
```

See Also

Open document, Select folder.

System Variables or 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 ({message}{; }{defaultPath}) → String

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
Function result	String	←	Access path to the selected folder

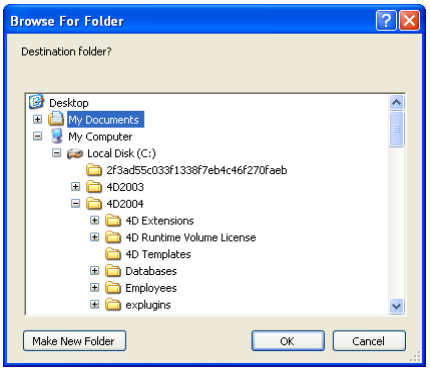
Description

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.

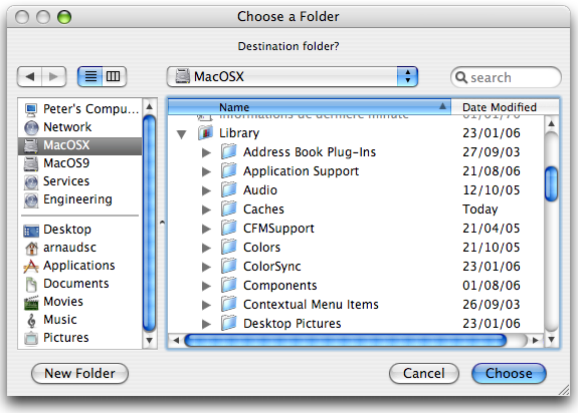
Note: This command does not modify 4D’s current folder.

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 "Select a destination folder":

Windows



Mac OS



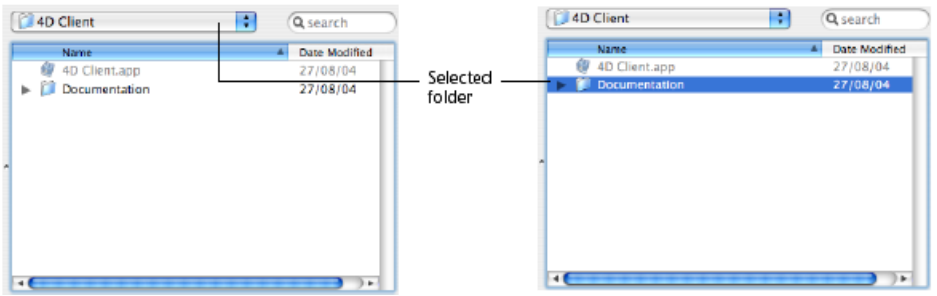
You can use the defaultPath parameter to provide a default folder location in the folder selection dialog box. You can pass three types of values in this parameter:
• The pathname of a valid folder using the syntax of the current platform.
• An empty string ("") to display the default user folder of 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 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))
    $vRef:=Create document($PictFolder+pictNames{$n};"PICT")
    If (OK=1)
        GET PICTURE FROM LIBRARY(pictRefs{$n};$vStoredPict)
        SAVE PICTURE TO FILE($vRef;$vStoredPict)
        CLOSE DOCUMENT($vRef)
    End if
End for
```

See Also

CREATE FOLDER, FOLDER LIST, Select document.

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.

See Also

Document creator, SET DOCUMENT PROPERTIES, SET DOCUMENT TYPE.

SET DOCUMENT POSITION (docRef; offset{; anchor})

Parameter	Type		Description
docRef	DocRef	→	Document reference number
offset	Number	→	File position (expressed in bytes)
anchor	Integer	→	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.

See Also

Get document position, RECEIVE PACKET, SEND PACKET.

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.

See Also

GET DOCUMENT PROPERTIES, SET DOCUMENT CREATOR, SET DOCUMENT TYPE.

SET DOCUMENT SIZE (document; size)

Parameter	Type		Description
document	DocRef	→	Document reference number
size	Number	→	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 document.

On Macintosh, the size of the document's data fork is changed.

See Also

Get document position, Get document size, SET DOCUMENT POSITION.

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")
```

See Also

Document type, MAP FILE TYPES, SET DOCUMENT CREATOR, SET DOCUMENT PROPERTIES.

SHOW ON DISK (pathname{; *})

Parameter	Type		Description
pathname	String	→	Pathname of item to show
*		→	If the item is a folder, show its contents

Description

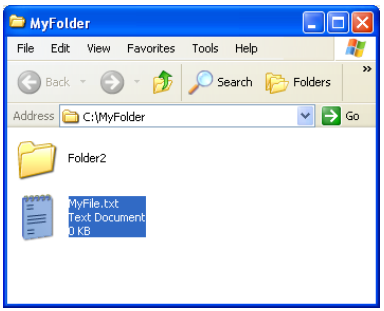
The SHOW ON DISK command displays the file or folder whose pathname was passed in the pathname parameter in a standard window of the operating system.
In a user interface, this command lets you designate the location of a specific file or folder.

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.

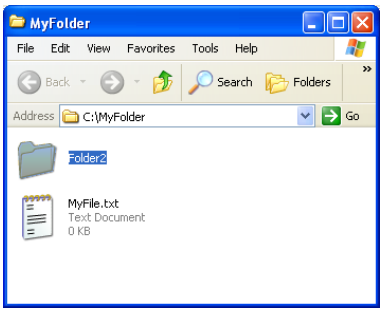
Examples

The following examples illustrate the operation of this command:

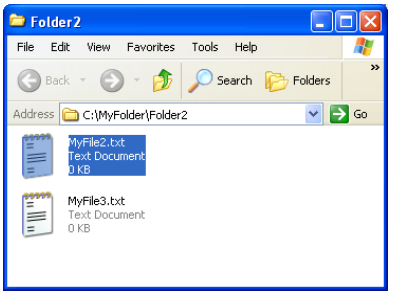
```
SHOW ON DISK("c:\\MyFolder\\MyFile.txt")  ` Displays the designated file
```



```
SHOW ON DISK("c:\\MyFolder\\Folder2")  ` Displays the designated folder
```




```
` Displays the contents of the designated folder
SHOW ON DISK("c:\\MyFolder\\Folder2";*)
```



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) → Number

Parameter	Type		Description
pathname	String	→	Pathname to directory, folder or document
Function result	Number	←	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 constant are provided by 4D:

Constant	Type	Value
Is a document	Long Integer	1
Is a directory	Long Integer	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
```

See Also

Create document, CREATE FOLDER.

VOLUME ATTRIBUTES (volume; size; used; free)

Parameter	Type		Description
volume	String	→	Volume name
size	Number	←	Volume size expressed in bytes
used	Number	←	Used space expressed in bytes
free	Number	←	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
```

See Also

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 or String 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.

On the other hand, on Windows, it returns the list of the volumes currently defined whether or not each volume is physically present (i.e. the volume A:\ will be returned whether or not a disk is actually present in the floppy drive).

Example

Using a scrollable area named asVolumes you want to display the list of the volumes defined or mounted on your machine, you write:

```
Case of
   : (Form event=On Load)
      ARRAY STRING(31;asVolumes;0)
      VOLUME LIST(asVolumes)

      ` ...
End case
```

See Also

DOCUMENT LIST, FOLDER LIST, VOLUME ATTRIBUTES.

53

System Environment

Count screens

Count screens → Longint

Parameter	Type		Description
This command does not require any parameters			
Function result	Longint	←	Number of monitors

Description

The Count screens command returns the number of screen monitors connected to your machine.

See Also

Menu bar screen, SCREEN COORDINATES, SCREEN DEPTH, Screen height, Screen width.

Current machine → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Network name of the machine

Description

The Current machine command returns the network name of your machine, as set in the Network Control Panel.

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
```

See Also

Current machine owner.

Current machine owner → String

Parameter	Type		Description
This command does not require any parameters			
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 Network Control Panel.

Example

See example for the command Current machine.

See Also

Current machine.

FONT LIST (fonts)

Parameter	Type		Description
fonts	Array	←	Array of font names

Description

The FONT LIST command populates the string or text array fonts with the names of the fonts available on your system.

Example

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 STRING(63;asFont;0)
      FONT LIST(asFont)
         ` ...

End case
```

See Also

Font name, Font number.

Font name (fontNumber) → String

Parameter	Type		Description
fontNumber	Longint	→	Font number for which to return the font name
Function result	String	←	Font name

Description

The Font name command returns the name of the font whose number is fontNumber. If there is no available font with that number, the command returns an empty string.

Examples

1. To display a form object with the default system font, you write:

```
FONT(myObject;Font name(0)) ` 0 is the font number of the default system font
```

2. To display a form object with the default application font, you write:

```
FONT(myObject;Font name(1)) ` 1 is the font number of the default application font
```

See Also

FONT LIST, Font number.

Font number (fontName) → Longint

Parameter	Type		Description
fontName	String	→	Font name for which to return the font number
Function result	Longint	←	Font number

Description

The Font number command returns the number of the font whose name is fontName. If there is no font with this name, the command returns 0.

Example

Some forms in your database use the font whose name is “Kind of Special.” Somewhere in your database, you could write:

```
If (Font number("Kind of Special")=0)
    ALERT("This form would look better if the font Kind of Special was installed.")
End if
```

See Also

FONT LIST, Font name.

Gestalt (selector; value) → Number

Parameter	Type		Description
selector	String	→	4-character gestalt selector
value	Number	←	Gestalt result
Function result	Number	←	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 unkown, 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

On Macintosh, using version 9.2 of Mac OS, the following code displays the alert “You're running system version 0x0920”:

```
$vlErrCode:=Gestalt("sysv";$vlInfo)
If ($vlErrCode=0)
    ALERT("You're running system version "+String($vlInfo;"&x"))
End if
```

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 (value)	Value(s) returned
Decimal separator (0)	Decimal separator (e.g.: ".")
Thousand separator (1)	Thousand separator (e.g.: ",")
Currency symbol (2)	Currency symbol (e.g.: "$")
System time short pattern (3)	Corresponding time display format in the form
System time medium pattern (4)	"HH:MM:SS"
System time long pattern (5)	
System date short pattern (6)	Corresponding date display format in the form
System date medium pattern (7)	"dddd d MMMM yyyy"
System date long pattern (8)	
Date separator (13)	Separator used in date formats (e.g.: "/")
Time separator (14)	Separator used in time formats (e.g.: ":")
Short date day position (15)	Position of the day, month, and year in the short date format:
Short date month position (16)	"1" = left
Short date year position (17)	"2" = middle
	"3" = right
System time AM label (18)	Additional label for a time before noon in 12-hour formats (e.g.: "Morning")
System time PM label (19)	Additional label for a time after noon in 12-hour formats (e.g.: "Afternoon")

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:

```
$MyFormat:="###"+GET SYSTEM FORMAT(Currency symbol) +"*******"+
                                    GET SYSTEM FORMAT(Decimal separator)+"**"
$Result:=String(amount;$MyFormat)
```

See Also

SET FORMAT.

LOG EVENT (message{; importance})

Parameter	Type		Description
message	String	→	Contents of the message
importance	Integer	→	Message's importance level

Note: This feature is only available on Windows.

Description

The LOG EVENT command allows you to add custom messages that will appear in the Windows **Log events**. This service maintains a log file that receives and stores messages coming from running applications. It therefore allows you to monitor the course of a worksession. For more information, please refer to the *4D Design Reference* manual.

For this feature to be available, Windows **Log events** service must be running.

Pass the message to write in the log events in message.

You can attribute a level of importance to message, which helps you to read and understand the log events. There are three levels of importance: Information, Warning, and Error. The importance parameter allows you to set the level of importance of the message.
4D provides you with the following predefined constants, placed in the "Windows Log Events" category:

Constant	Type	Value
Information Message	Integer	0
Warning Message	Integer	1
Error Message	Integer	2

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, you could write the following line of code in the On Startup Database Method:

```
LOG EVENT ("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 → Longint

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

HIDE MENU BAR, Menu bar screen, SHOW MENU BAR.

Menu bar screen → Longint

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

Count screens, Menu bar height.

PLATFORM PROPERTIES (platform{; system{; processor{; language}}})

Parameter	Type		Description
platform	Number	←	2 = Mac OS, 3 = Windows
system	Number	←	Depends on the version you are running
processor	Number	←	Processor family
language	Number	←	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	Long Integer	2
Windows	Long Integer	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 = 8  → you are under Mac OS 8.x
    `If $vlResult = 9  → you are under Mac OS 9.x
    `If $vlResult = 16  → you are under Mac OS 10.x
```

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 Windows 95 or Windows 98 (both obsolete).

Note: The high level bit fixes the sign of the long integer value. Therefore, in 4D, you just need to test the sign of the value; if it is positive you are running Windows NT, Windows 2000, Windows XP or Windows Vista. You can also use the Bitwise operators.

The low byte gives the major Windows version number. If it returns 4, you are running Windows 95, 98 or Windows NT 4. If it returns 5, you are running Windows 2000 or Windows XP (in both cases, the sign of the value tells whether or not you are running NT/2000). If it returns 6, you are running Windows Vista..

The next low byte gives the minor Windows version number. If you are running Windows 95, this value is 0.

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	Long Integer	586
Power PC	Long Integer	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.

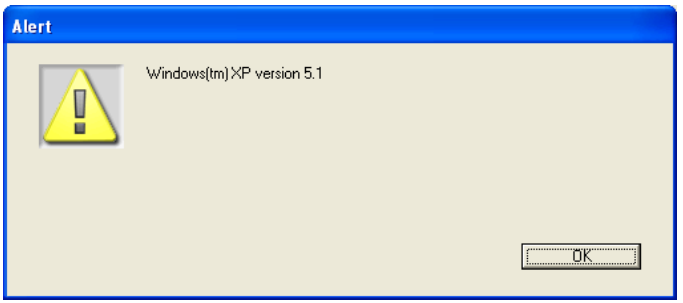
Example

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)
         $winMajVers:=((2^31)+$vlSystem)%256
         $winMinVers:=(((2^31)+$vlSystem)\256)%256
         If ($winMinVers=0)
            $vsPlatformOS:="Windows™ 95"
         Else
            $vsPlatformOS:="Windows™ 98"
         End if
      Else
         $winMajVers:=$vlSystem%256
         $winMinVers:=($vlSystem\256)%256
         Case of
            : ($winMajVers=4)
               $vsPlatformOS:="Windows™ NT"
```

```
            : ($winMajVers=5)
                If ($winMinVers=0)
                    $vsPlatformOS:="Windows™ 2000"
                Else
                    $vsPlatformOS:="Windows™ XP"
                End if
            : ($winMajVers=6)
                $vsPlatformOS:="Windows™ Vista"

            End case
        End if
        $vsPlatformOS:=$vsPlatformOS+" version "+String($winMajVers)+"."+
                                                        String($winMinVers)
    Else
        $vsPlatformOS:="Mac OS™ version "
        If (($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 Macintosh, you get an alert box similar to this:

See Also

Bitwise Operators.

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.

See Also

Count screens, Menu bar screen, SCREEN DEPTH.

SCREEN DEPTH (depth; color{; screen})

Parameter	Type		Description
depth	Number	←	Depth of the screen (number of colors = 2 ^ depth)
color	Number	←	1 = Color screen, 0 = Black and white or Gray scale
screen	Number	→	Screen number, or main screen if omitted

Description

The Screen depth command returns in depth and color information about the monitor.

After the call:

• 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	Long Integer	0
Four colors	Long Integer	2
Sixteen colors	Long Integer	4
Two fifty six colors	Long Integer	8
Thousands of colors	Long Integer	16
Millions of colors 24 bit	Long Integer	24
Millions of colors 32 bit	Long Integer	32

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 gray scale	Long Integer	0
Is color	Long Integer	1

• 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
```

See Also

Count screens, SET SCREEN DEPTH.

Screen height {(*)} → Number

Parameter	Type		Description
*	Number	→	Windows: height of application window, or height of screen if * is specified Macintosh: height of main screen
Function result	Number	←	Height expressed in pixels

Description

On Windows, Screen height returns the height of 4D application window (MDI window). If you specify the optional * parameter, Screen height 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.

See Also

SCREEN COORDINATES, Screen width.

Screen width {(*)} → Number

Parameter	Type		Description
*	Number	→	Windows: width of application window, or width of screen if * is specified Macintosh: width of main screen
Function result	Number	←	Width expressed in pixels

Description

On Windows, Screen width returns the width of 4D application window (MDI window). If you specify the optional * parameter, Screen width 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.

See Also

SCREEN COORDINATES, Screen height.

Select RGB Color ({defaultColor}{; }{message}) → Longint

Parameter	Type		Description
defaultColor	Longint	→	Preselected RGB color
message	Alpha 255	→	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:

Windows

Macintosh

The optional defaultColor parameter can be used to preselect 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 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 can be used to customize the title of the system window. By default, if this parameter is omitted, the title “Colors” is displayed.

If the user validates the dialog box, 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.

Note: This command must not be executed on the server machine nor within a Web process.

See Also

SET RGB COLORS.

System Variables or Sets

If the user validates the dialog box, the system variable OK is set to 1; otherwise, it is set to 0.

SET SCREEN DEPTH (depth{; color{; screen}})

Parameter	Type		Description
depth	Number	→	Depth of the screen (number of colors = 2 ^ Screen depth)
color	Number	→	1 = Color, 0 = Gray Scale
screen	Number	→	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.

See Also

SCREEN DEPTH.

System folder {(type)} → String

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
System	Long Integer	0
Fonts	Long Integer	1
Preferences or Profiles_All	Long Integer	2
Preferences or Profiles_User	Long Integer	3
Startup Items_All	Long Integer	4 (obsolete under Mac OS, see note 1)
Startup Items_User	Long Integer	5 (obsolete under Mac OS, see note 1)
Mac Shutdown Items_All	Long Integer	6 (obsolete, see note 1)
Mac Shutdown Items_User	Long Integer	7 (obsolete, see note 1)
Apple or Start Menu_All	Long Integer	8 (obsolete under Mac OS, see note 1)
Apple or Start Menu_User	Long Integer	9 (obsolete under Mac OS, see note 1)
Mac Extensions	Long Integer	10 (obsolete, see note 1)
Mac Control Panels	Long Integer	11 (obsolete, see note 1)
System Win	Long Integer	12
System32 Win	Long Integer	13
Favorites Win	Long Integer	14
Desktop Win	Long Integer	15
Program Files Win	Long Integer	16

Notes:
• The folders corresponding to constants 4 to 11 no longer exist under Mac OS X (they were used under Mac OS 9 only). When these constants are used under Mac OS, System folder returns an empty string.
Constants 6, 7, 10 and 11 (Mac OS only) are thus totally obsolete starting with version 2004 of 4D. Constants 4, 5, 8 and 9 however can still be used under Windows.
• The constants System Win, System32 Win, Favorites Win, Desktop Win and Program Files 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).

See Also

Get 4D folder, Temporary folder.

Temporary folder → String

Parameter	Type		Description
This command does not require any parameters			
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 command APPEND DATA TO PASTEBOARD.

See Also

System folder.

54

Table

Current default table

version 3

Current default table → Pointer

Parameter	Type		Description
This command does not require any parameters			
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))
```

See Also

DEFAULT TABLE, Table, Table name.

Current form table

version 11 (Modified)

Current form table → Pointer

Parameter	Type		Description
This command does not require any parameters			
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
```

See Also

DIALOG, INPUT FORM, OUTPUT FORM, PRINT SELECTION.

DEFAULT TABLE (aTable)

Parameter	Type		Description
aTable	Table	→	Table to set as the default

Description

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:

```
INPUT FORM ([Table]; "form")
```

If the default table is first set to [Table], the same command could be written this way:

```
INPUT FORM ("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.

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.

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 INPUT FORM and ADD RECORD commands both require a table as the first parameter:

```
INPUT FORM ([Customers];"Add Recs")
Repeat
    ADD RECORD ([Customers])
Until (OK = 0)
```

Specifying the default table results in this code:

```
DEFAULT TABLE ([Customers])
INPUT FORM ("Add Recs")
Repeat
    ADD RECORD
Until (OK = 0)
```

See Also

Current default table, NO DEFAULT TABLE.

NO DEFAULT TABLE

Parameter	Type	Description

This command 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 GET FORM 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
```

See Also

DEFAULT TABLE.

55

Tools

Choose (criterion; value{; value2; ...; valueN}) → Expression

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.

Examples

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
```

2. Voici une utilisation type de la command avec un critère Number :

```
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
```

DECODE (blob)

Parameter	Type		Description
blob	BLOB	→	BLOB encoded in Base64 format
		←	Decoded BLOB

Description

The DECODE command allows you to decode the BLOB coded in Base64 format passed in the blob parameter. The command directly modifies the BLOB passed as a parameter.

The command does not verify the contents of the blob. You must verify that the data passed is really coded in Base64 format, otherwise the result will be incorrect.

See Also

ENCODE.

ENCODE (blob)

Parameter	Type		Description
blob	BLOB	→	BLOB to encode in Base64 format
		←	BLOB encoded in Base64 format

Description

The ENCODE command encodes the BLOB passed in the blob parameter in Base64 format. 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.

See Also

DECODE.

GET MACRO PARAMETER

version 11

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 (fileName{; inputStream{; outputStream{; errorStream}}})

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)

Warning: This command is designed for advanced users.

Description

The LAUNCH EXTERNAL PROCESS command allows you to launch 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.

Note: For 4D Pack users, this command has the same functions (plus expanded features) as the AP_Sublaunch command.

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 manage text data of a size greater than 32 KB or 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.

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 preferableb 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 /Cstart \"\" \""+document+"\"")
End if
```

See Also

SET ENVIRONMENT VARIABLE.

System Variables or 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.

SET DICTIONARY (dictionary)

Parameter	Type		Description
dictionary	Longint	→	Dictionary to use for spell-check

Description

The SET 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). The modification of the current dictionary is reflected in all the processes of the database for the session.

By default, 4D uses the dictionary corresponding to the application language. Five main dictionaries are available: English, French, German, Spanish and Norwegian.

In dictionary, pass the number of the dictionary to use. You can use one of the following predefined constants, which are found in the "Dictionaries" theme:

Constant	Type	Value
English Dictionary	Longint	69632
German Dictionary	Longint	131584
Spanish Dictionary	Longint	196608
French Dictionary	Longint	262144
Norwegian Dictionary	Longint	589824

In addition, numerous variants are available for each of the four main languages. Here is the full list of all variants supported by the command. To load a variant, pass its value directly in the dictionary parameter:

Dictionary	Value
English (United Kingdom)	65536
English Irish (Ireland)	65600
English Australian (Australia)	65664
English of New Zealand	65680
English American (USA)	65792
English Canadian (Canada)	65920
English South African (South Africa)	66048
English West Indian (Caribbean)	66176
English Jamaican (Jamaica)	66192
English (United Kingdom + America)	69632 (*)

German standard (Germany, old spelling)	131072
German of Luxembourg	131073
German of Austria	131088
German of Liechtenstein	131089
German of Switzerland (old spelling)	131104
German of South Tyrol	131120
German New spelling	131328
German of Switzerland New spelling	131360
German Old and New spelling	131584 (*)
German of Switzerland Old and New spelling	131616
Spanish standard (Spain)	196608 (*)
Spanish of Latin America standard	196864
Spanish Argentinean (Argentina)	196865
Spanish Bolivian (Bolivia)	196866
Spanish Chilean (Chile)	196867
Spanish Columbian (Columbia)	196868
Spanish Cuban (Cuba)	196869
Spanish Costa Rican (Costa Rica)	196870
Spanish Dominican (Dominican Rep.)	196871
Spanish Ecuadorian (Ecuador)	196872
Spanish Guatemalan (Guatemala)	196873
Spanish Honduran (Honduras)	196874
Spanish Mexican (Mexico)	196875
Spanish Nicaraguan (Nicaragua)	196876
Spanish Panamanian (Panama)	196877
Spanish Paraguayan (Paraguay)	196878
Spanish Peruvian (Peru)	196879
Spanish Puerto Rican (Puerto Rico)	196880
Spanish Salvadorian (El Salvador)	196881
Spanish Uruguayan (Uruguay)	196882
Spanish Venezuelan (Venezuela)	196883
Spanish Guinean (Equatorial Guinea)	197121
France, Monaco, Valle d'Aosta	262144 (*)
Canada	262160
Louisiana	262161
Belgium	262176
Luxembourg	262177
Switzerland	262192
Martinique, Guadeloupe, Haïti, Guyana	262208
Reunion, Seychelles, Comoro, Mauritius	262224
Tahiti, New Caledonia, Vanuatu, etc.	262240
Morocco, Algeria, Tunisia	262256
French African standard	262272
Benin	262273
Burkina Faso	262274
Burundi	262275
Cameroon	262276

Central African Republics	262277
Congo (Brazzaville)	262278
Democratic Republic of Congo (ex-Zaire)	262279
Ivory Coast	262280
Djibouti	262281
Gabon	262282
Guinea	262283
Mauritania	262284
Niger	262285
Rwanda	262286
Senegal	262287
Chad	262288
Togo	262289
Bokmal Norwegian	589824 (*)
Nynorsk Norwegian	590080
Samnorsk Norwegian	590336

(*) standard dictionary that is installed when you use a constant.

Note: The Norwegian dictionary is not present by default in 4D. Please contact 4D in order to obtain it free of charge. You must then install it in the 4D Extensions/Spellcheck folder.

See Also

SPELL CHECKING.

System Variables or 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.

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 SET CGI EXECUTABLE or LAUNCH EXTERNAL PROCESS commands.

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 SET CGI EXECUTABLE command, please refer to the Using CGIs section in the “Web Server” chapter.
• 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.
These variables are valid in the current process for the next call to LAUNCH EXTERNAL PROCESS.

Example

Refer to examples 7 and 8 of the LAUNCH EXTERNAL PROCESS command.

See Also

LAUNCH EXTERNAL PROCESS, SET CGI EXECUTABLE, Using CGIs.

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.

Compatibility 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:=""
    ` Select all ([Customers])
$output_text:=$output_text+Command name(47)+"("+$input_text+")"
    ` $i:=Records in selection([Customers])
$output_text:=$output_text+"$i:="+Command name(76)+"("+ $input_text+")"
SET MACRO PARAMETER(Highlighted method text;$output_text)
    `Replaces the selected text by the new code
```

See Also

GET MACRO PARAMETER.

SPELL CHECKING

Parameter	Type	Description

This command does not require any parameters

Description

The SPELL CHECKING command provokes 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.

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 SET DICTIONARY command.

See Also

SET DICTIONARY.

56

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.

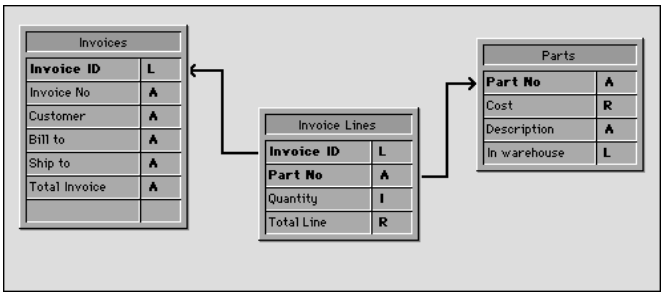
Starting with version 11, 4D supports nested transactions, i.e. transactions on several hierarchical levels. The number of subtransactions allowed is unlimited. The Transaction level command can be used to find out the current transaction level where the code is executed. When you use nested transactions, 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.

Compatibility option
Since nested transactions can lead to malfunctioning in databases developed with previous versions of 4D, it is disabled by default in converted databases (transactions remain limited to a single level). If you want to take advantage of transactions on several levels in a converted database, you must indicate this explicitly by checking the **Allow Nested Transactions** option on the “Application/Compatibility” page of the application Preferences.
This option only appears in converted databases. By default, it is not checked. It is specific to each database. It has no effect on transactions carried out in the SQL engine of 4D. SQL transactions are always multi-level.

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])
INPUT FORM([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])
INPUT FORM([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
                ` Try getting the record in Read/Write access
            While (Locked([Parts]) & (OK=1))
                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.

See Also

CANCEL TRANSACTION, In transaction, START TRANSACTION, VALIDATE TRANSACTION.

CANCEL TRANSACTION

Parameter	Type	Description

This command 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 during the transaction.

See Also

In transaction, START TRANSACTION, Transaction level, Using Transactions, VALIDATE TRANSACTION.

In transaction

In transaction → Boolean

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

CANCEL TRANSACTION, START TRANSACTION, Triggers, VALIDATE TRANSACTION.

START TRANSACTION

Parameter	Type	Description

This command 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.

See Also

CANCEL TRANSACTION, In transaction, Transaction level, Using Transactions, VALIDATE TRANSACTION.

Transaction level

Transaction level → Longint

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

In transaction, START TRANSACTION, Using Transactions.

VALIDATE TRANSACTION

Parameter	Type	Description
This command 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. VALIDATE TRANSACTION 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.

See Also

CANCEL TRANSACTION, In transaction, START TRANSACTION, Using Transactions.

57

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, modify, and load). 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

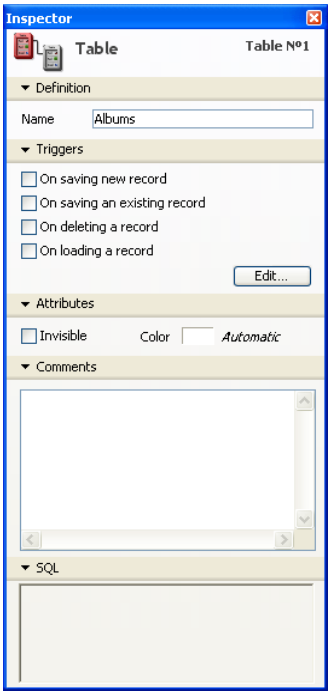
By default, when you create a table in the Design Environment, it has no trigger.

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 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 or ADD RECORD 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.

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 or MODIFY RECORD 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.

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 or DELETE SELECTION).
• 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.

On loading a record
If this option is selected, the trigger will be invoked each time a record of the table is loaded. This includes all situations in which a current record is loaded from the data file. You will use this option less often than the three previous ones.

In order to optimize the operation of 4D, the On loading a record option never triggers a call to the trigger when using a command that can take advantage of the index.

In fact, when the index is used, records are not loaded and conversely, if the index is not used (i.e., if the field being processed is not indexed), records are loaded. That would mean that a trigger for which the On loading a record option was selected could either be or not be executed depending on the Indexed attribute for the processed field. Rather than keeping a behavior that is difficult to anticipate, the decision was made to never execute the trigger with the On loading a record option selected when using a command that could take advantage of the index.

Note: If the On loading a record option is selected, the trigger will be executed when a current record is loaded from the data file, except for the following functions:

• Queries: User queries that were prepared in the standard query editor or by using the QUERY or QUERY SELECTION commands.
• Order by: Sorts that were prepared in the standard Order by Editor or by using the ORDER BY command.
• On a series: Sum, Average, Min, Max, Std deviation, Variance, Sum square.
• Commands: RELATE ONE SELECTION, RELATE MANY SELECTION.

IMPORTANT: If you execute an operation or call a command that acts on multiple records, the trigger is called once for each record. For example, if you call APPLY TO SELECTION for a table whose current selection is composed of 100 records, the trigger will be invoked 100 times.

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 four **database events** described above. Within the trigger, you detect which event is occurring by calling the Database 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 Database event. You can use the constants of the Database Events theme:

```
    ` Trigger for [anyTable]
C_LONGINT($0)
$0:=0  ` Assume the database request will be granted
Case of
   : (Database event=On Saving New Record Event)
      ` Perform appropriate actions for the saving of a newly created record
   : (Database event=On Saving Existing Record Event)
      ` Perform appropriate actions for the saving of an already existing record
   : (Database event=On Deleting Record Event)
      ` Perform appropriate actions for the deletion of a record
   : (Database event=On Loading Record Event)
      ` Perform appropriate actions for the loading into memory 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, or just after it has been loaded.
• 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
    : (Database event=On Saving New Record Event)
        [Documents]Creation Stamp:=Time stamp
        [Documents]Modification Stamp:=Time stamp
    : (Database 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
$dbEvent:=Database event
Case of
    : (($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, 4D Open client with 4D Server) 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.

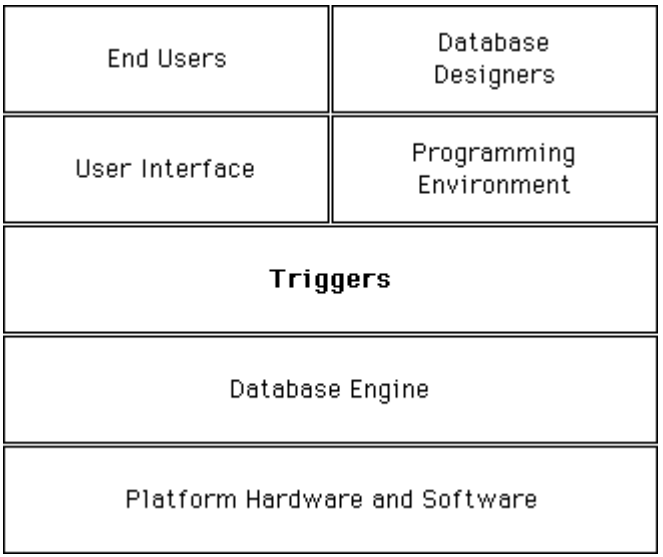
At the process level, you handle trigger errors the same way you handle database engine errors:
• You can let 4D display the standard error dialog box, then the method is halted.
• You can use an error-handling method installed using ON ERR CALL and recover the error the appropriate way.

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, 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**.

In particular, the trigger works with the current selections, current records, table read/write states, and record locking operations of the invoking process.

Warning: A trigger cannot and must not change the current record of the table to which it is attached. Within a trigger, if you need to check a unique value on multiple fields, use the SET QUERY DESTINATION command, which allows you to query a table without changing the current selection or current record of the table.

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**: An independent process variables table is shared by all the triggers. 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.
- **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 Debugger 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.

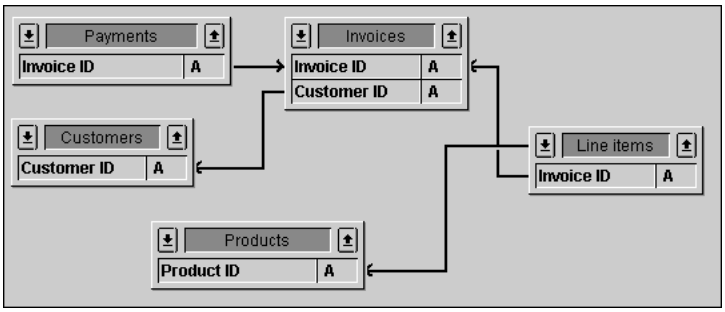
Triggers and Transactions

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.

Using Sequence Numbers within a Trigger

While handling an On Saving New Record Event database event, you can call the Sequence number command to maintain a unique ID number for the records of a table.

Example

```
      ` Trigger for table [Invoices]
   Case of
      : (Database event=On Saving New Record Event)
            ` ...
         [Invoices]Invoice ID Number:=Sequence number ([Invoices])
            ` ...
   End case
```

See Also

Database event, Methods, Record number, Trigger level, TRIGGER PROPERTIES.

Database event

Database event → Longint

Parameter	Type			Description
This command does not require any parameters				
Function result	Longint	←	0	Outside any trigger execution cycle
			1	Saving a new record
			2	Saving an existing record
			3	Deleting a record
			4	Loading a record

Description

Called from within a trigger, the Database 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:

Constant	Type	Value
On Saving New Record Event	Long Integer	1
On Saving Existing Record Event	Long Integer	2
On Deleting Record Event	Long Integer	3
On Loading Record Event	Long Integer	4

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 command Database event to structure your triggers as follows:

```
    ` Trigger for [anyTable]
C_LONGINT($0)
$0:=0   ` Assume the database request will be granted
Case of
    : (Database event=On Saving New Record Event)
        ` Perform appropriates action for the saving of a newly created record
    : (Database event=On Saving Existing Record Event)
        ` Perform appropriates actions for the saving of an already existing record
    : (Database event=On Deleting Record Event)
        ` Perform appropriates actions for the deletion of a record
    : (Database event=On Loading Record Event)
        ` Perform appropriates actions for the loading into memory of a record
End case
```

See Also

In transaction, Trigger level, TRIGGER PROPERTIES, Triggers.

Trigger level

Trigger level → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	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 in the section Triggers.

See Also

Database event, TRIGGER PROPERTIES, Triggers.

TRIGGER PROPERTIES

TRIGGER PROPERTIES (triggerLevel; dbEvent; tableNum; recordNum)

Parameter	Type		Description
triggerLevel	Number	→	Trigger execution cycle level
dbEvent	Number	←	Database event
tableNum	Number	←	Involved table number
recordNum	Number	←	Involved record number

Description

The TRIGGER PROPERTIES command returns information about the trigger execution level you pass in triggerLevel. You use TRIGGER PROPERTIES in conjunction with Trigger level to perform different actions depending on the cascading of trigger execution levels. For more information, see the topic Cascading Triggers in the section 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:

Constant	Type	Value
On Saving New Record Event	Long Integer	1
On Saving Existing Record Event	Long Integer	2
On Deleting Record Event	Long Integer	3
On Loading Record Event	Long Integer	4

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.

Note: Remember that while in transaction, newly created records have temporary record numbers.

See Also

About Record Numbers, Database event, Trigger level, Triggers.

58

User Forms

Overview of user forms

In 4D version 2004, 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 INPUT FORM and OUTPUT FORM 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 Developer, 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 new LIST USER FORMS command allows finding valid user forms at any time). It is also in this file that the INPUT FORM and OUTPUT FORM 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 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 (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.

See Also

DELETE USER FORM, EDIT FORM, INPUT FORM, LIST USER FORMS, OUTPUT FORM, Overview of user forms.

System Variables or Sets

The OK variable returns 1 if the operation is executed properly; otherwise, it returns 0.

Error Handling

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 (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.

If the user form was created directly using the EDIT FORM command, pass an empty string ("") in userForm.

See Also

CREATE USER FORM, LIST USER FORMS, Overview of user forms.

System Variables or Sets

If the user form is properly removed, the OK variable returns 1. Otherwise, OK is set to 0.

Error Handling

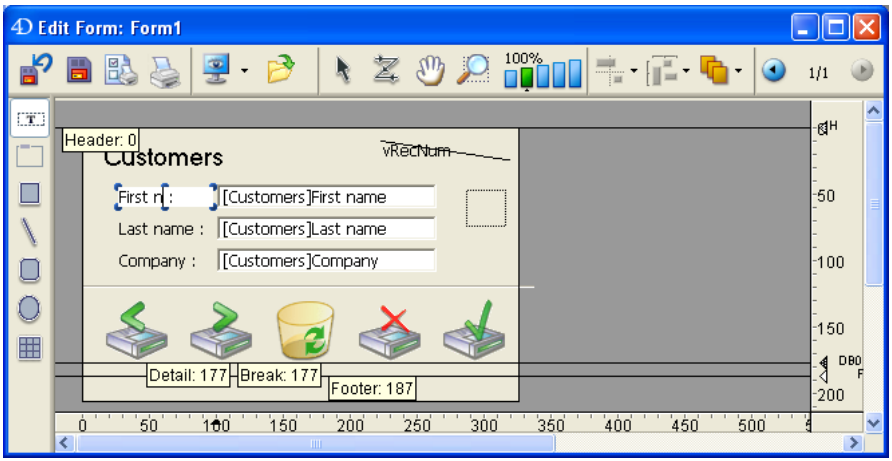
An error is generated if:
• the user form does not exist,
• the user does not have the proper access to remove the user form.
You can intercept this error with the error-handling method installed by the ON ERR CALL command.

EDIT FORM (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 aTable, 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 allows setting 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 **Extras** 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
```

See Also

CREATE USER FORM, DELETE USER FORM, LIST USER FORMS, Overview of user forms.

System Variables or 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 Handling

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 (aTable; form; userFormArray)

Parameter	Type		Description
aTable	Table	→	Source form table
form	String	→	Source table form name
userFormArray	Array string	←	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.

See Also

CREATE USER FORM, EDIT FORM, Overview of user forms.

59

User Interface

BEEP

Parameter	Type	Description

This command 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
```

See Also

PLAY.

Caps lock down

Caps lock down → Boolean

Parameter	Type		Description
This command does not require any parameters			
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 command Shift down.

See Also

Macintosh command down, Macintosh control down, Macintosh option down, Shift down, Windows Alt down, Windows Ctrl down.

Focus object → Pointer

Parameter	Type		Description
This command does not require any parameters			
Function result	Pointer	←	Pointer to the object having the focus

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 an array type list box, the Focus object function returns a pointer to the column (the array) of the list box that has the focus. In the case of list boxes of the selection type, the 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, a pointer to the variable of the list box.

This command cannot be used with fields in subforms.

Note: This command is to be used only in 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
      ...
      ` If it is a string or text area
   : ((Type ($vp->) = Is Alpha field) | (Type($vp->) = Is String var))
   $vp-> := Uppercase ($vp->)  ` Change the area to uppercase
End case
```

GET FIELD TITLES (table; fieldTitles; fieldNums)

Parameter	Type		Description
table	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 table. 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.

See Also

GET TABLE TITLES, SET FIELD TITLES.

GET HIGHLIGHT (area; startSel; endSel)

Parameter	Type		Description
area	Field | Variable	→	Enterable field or variable
startSel	Number	←	Current text selection starting position
endSel	Number	←	Current text selection ending position

Description

The GET HIGHLIGHT command is used to determine what text is currently highlighted.

Warning: Although you pass a enterable field or variable name to GET HIGHLIGHT, this command returns a significant selection position only when it is applied to the area currently being edited.

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.

Examples

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
```

2. See example for the command FILTER KEYSTROKE.

See Also

FILTER KEYSTROKE, HIGHLIGHT TEXT, Keystroke.

GET MOUSE (mouseX; mouseY; mouseButton{; *})

Parameter	Type		Description
mouseX	Number	←	Horizontal coordinate of mouse
mouseY	Number	←	Vertical coordinate of mouse
mouseButton	Number	←	Mouse button state: 0 = Button up 1 = Button down 2 = Right button down 3 = Both buttons down
*		→	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 horizonal 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 frontmost window 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 command Pop up menu.

See Also

Caps lock down, Macintosh command down, Macintosh control down, Macintosh option down, ON EVENT CALL, Shift down, Windows Alt down, Windows Ctrl down.

version 2004 (Modified)

Compatibility Note
This command is maintained only for compatibility reasons. In new databases (created with 4D starting with version 2004), the platform interface is managed automatically by the program and this command is ignored. It can still be used in converted database, although it is recommended to activate the new interface management feature in the Preferences dialog box ("System" option).

Get platform interface → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Current platform interface in use

Description

The Get platform interface command returns a numeric value that denotes the current platform interface used for displaying forms.

The function can return one of the following values:

Constant	Type	Value
Automatic Platform	Longint	-1
Mac OS 7	Longint	0
Windows 3.11, NT 3.51	Longint	1
Windows 9x	Longint	2
Mac OS 9	Longint	3
Mac Theme	Longint	4

You can change the platform interface using the command SET PLATFORM INTERFACE or within the **Preferences** dialog box.

See Also

SET PLATFORM INTERFACE.

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.

See Also

GET FIELD TITLES, SET TABLE TITLES.

HIDE MENU BAR

Parameter	Type	Description

This command 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)
INPUT FORM([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.

See Also

HIDE TOOL BAR, SHOW MENU BAR, SHOW TOOL BAR.

HIDE TOOL BAR

Parameter	Type	Description
This command does not require any parameters		

Description

The HIDE TOOL BAR command makes the toolbar invisible.

If the toolbar was already hidden, HIDE TOOL BAR does nothing.

See Also

HIDE MENU BAR, SHOW MENU BAR, SHOW TOOL BAR.

HIGHLIGHT TEXT (area; startSel; endSel)

Parameter	Type		Description
area	Field | Variable	→	Enterable field or variable
startSel	Number	→	New text selection starting position
endSel	Number	→	New text selection ending position

Description

The HIGHLIGHT TEXT command highlights a section of the text in area.

If area is not the object currently being edited, the focus is then set to this area.

Note: This command cannot be used with fields in the List form of 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 area, then all characters between startSel and the end of the text are highlighted.

Examples

1. The following example selects all the characters of the enterable field [Products]Comments:

```
HIGHLIGHT TEXT([Products]Comments;1;Length([Products]Comments)+1)
```

2. The following example moves the insertion point to the beginning of the enterable field [Products]Comments:

```
HIGHLIGHT TEXT([Products]Comments;1;1)
```

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)
```

4. See example for the command FILTER KEYSTROKE.

See Also

GET HIGHLIGHT.

INVERT BACKGROUND ({*; }textVar | textField)

Parameter	Type		Description
*		→	Allows entry of a variable or object name
textVar | textField	Variable | Field	→	Text variable or field to invert

Description

INVERT BACKGROUND is used to invert textVar or textField in the form.

The scope of the command is the form being used.

You can use INVERT BACKGROUND when displaying on screen or printing to a dot matrix printer. A postscript printer will not print an inverted background.

You cannot invert a variable in an output form. Avoid using INVERT BACKGROUND on an enterable variable. Entering characters will only partially erase the inverted display.

Example

This example is an object method for a variable in an input form. It tests the value of a field. If the field is positive, the object method does nothing. If the field is negative, the object method inverts the display of the variable in the form:

```
vAmount:=[Accounts]Amount  ` Put the value of field in the variable
If (vAmount < 0)  ` If it is a negative amount…
    INVERT BACKGROUND (vAmount)  ` Invert the background
End if
```

Note: This command, originally created for black and white user interfaces, is now rarely used. You now generally use colors to highlight a field or a variable.

See Also

SET COLOR, SET RGB COLORS.

Macintosh command down

Macintosh command down → Boolean

Parameter	Type		Description
This command does not require any parameters			
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 command Shift down.

See Also

Caps lock down, Macintosh control down, Macintosh option down, Shift down, Windows Alt down, Windows Ctrl down.

Macintosh control down → Boolean

Parameter	Type		Description
This command does not require any parameters			
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 command Shift down.

See Also

Caps lock down, Macintosh command down, Macintosh option down, Shift down, Windows Alt down, Windows Ctrl down.

Macintosh option down

Macintosh option down → Boolean

Parameter	Type		Description
This command does not require any parameters			
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 command Shift down.

See Also

Caps lock down, Macintosh command down, Macintosh control down, Shift down, Windows Alt down, Windows Ctrl down.

PLAY (objectName{; channel})

Parameter	Type		Description
objectName	String	→	Sound name Windows: .WAV, .MID or .AVI file Any platform: Mac OS-based 'snd' resource or empty string for stopping asynchronous play
channel	Number	→	if specified, synthesizer channel and asynchronous if omitted, synchronous

Description on Windows

On Windows, the PLAY command plays sound (.WAV files), MIDI (.MID files), or Video (.AVI files) Windows files. You pass the full pathname of the file you want to play in objectName.

Note: You cannot play multimedia files or objects in asynchronous mode. To do so, use OLE Services.

On Macintosh or on Windows (with some restrictions, see Important Note below), the command PLAY plays the sound resource named by objectName on Macintosh.

The channel parameter specifies the Macintosh synthesizer channel. If channel is not specified, the channel is for simple digitized sounds and is synchronous. Synchronous means that all processing stops until the sound has finished. If channel is 0, the channel is for simple digitized sounds and is asynchronous. Asynchronous means that processing does not stop and the sound plays in the background.

To stop playing a synchronous sound, use the following statement:

```
PLAY ("";0)
```

If you work with a database on Macintosh and Windows concurrently, you can also play Macintosh sounds on the Windows platform. To do so:

• On the Macintosh, using a resource editor such as ResEdit or Resorcerer, copy the required 'snd ' resources into the resource fork of the structure file.

• Transport the database from Macintosh to Windows, using 4D Transporter.

Important Note: The Windows version of 4D does not play Macintosh sounds that have been compressed by MACE. If your Macintosh 'snd' resource does not play on Windows, determine whether it complies with the following requirements:

snd resource field	**Value (in hexadecimal)**
Version	0x0001
NbSynth	0x0001
SynthResID	0x0005
SynthInitOptions	0x000000A0
NbSoundCommand	0x0001
FirstCommand	0x8051

You can check the internal data of a 'snd' resource using Resorcerer.

Examples

1. The following example shows how to play a video file on Windows:

```
$DocRef := Open document ( ""; "AVI")
If (OK=1)
    CLOSE DOCUMENT($DocRef)
    PLAY (Document)
End if
```

2. The following example code appears in a startup method. It welcomes the user with a sound called Welcome Sound on Macintosh:

```
PLAY ("Welcome Sound")  ` Play the Welcome Sound
```

See Also

BEEP.

Pop up menu (contents{; default{; xCoord; yCoord}}) → Number

Parameter	Type		Description
contents	Text	→	Menu text definition
default	Number	→	Number of menu item selected by default
xCoord	Number	→	X coordinate of upper left corner
yCoord	Number	→	Y coordinate of upper left corner
Function result	Number	←	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 "(-" 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 allows you to specify 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 are used to 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 useful in particular 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:

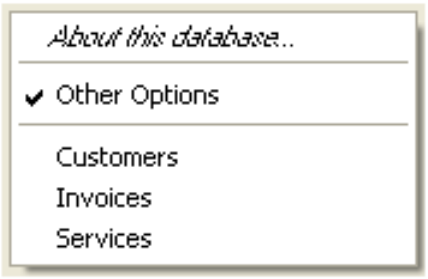
```
    ` MY SPEED MENU project method
GET MOUSE($vlMouseX;$vlMouseY;$vlButton)
If (Macintosh control down | ($vlButton=2))
    $vtItems:="About this database...<l;(-;!-Other Options;(-"
    For ($vlTable;1;Get last table number)
        If(Is table number valid($vlTable))
        $vtItems:=$vtItems+";"+Table 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
```

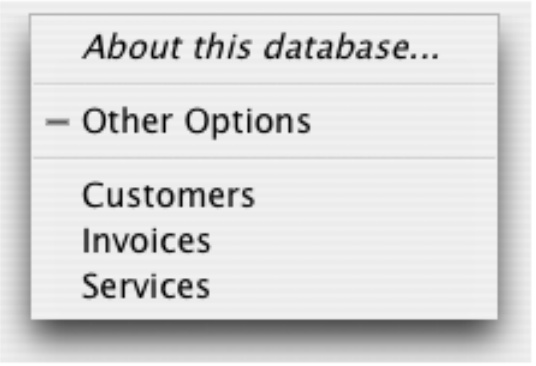
This project method can be called from:
• The method of a form object that reacts to a mouse click without waiting for the mouse button to be released (i.e., an invisible button)
• A process that “spies” events and communicate with the other processes
• An event-handling method installed using ON EVENT 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.





See Also

Dynamic pop up menu, GET MOUSE.

POST CLICK (mouseX; mouseY{; process}{; *})

Parameter	Type		Description
mouseX	Number	→	Horizontal coordinate
mouseY	Number	→	Vertical coordinate
process	Number	→	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.

See Also

POST EVENT, POST KEY.

POST EVENT (what; message; when; mouseX; mouseY; modifiers{; process})

Parameter	Type		Description
what	Number	→	Type of event
message	Number	→	Event message
when	Number	→	Event time expressed in ticks
mouseX	Number	→	Horizontal coordinate of mouse
mouseY	Number	→	Vertical coordinate of mouse
modifiers	Number	→	Modifier keys state
process	Number	→	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
Mouse down event	Long Integer	1
Mouse up event	Long Integer	2
Key down event	Long Integer	3
Key up event	Long Integer	4
Auto key event	Long Integer	5

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 Events (modifiers) constants. 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.

See Also

POST CLICK, POST KEY.

POST KEY (code{; modifiers{; process}})

Parameter	Type		Description
code	Number	→	Character code or function key code
modifiers	Number	→	State of modifier keys
process	Number	→	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. 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 command Process number.

See Also

POST CLICK, POST EVENT.

REDRAW (object)

Parameter	Type		Description
object	Object	→	Subtable for which to redraw the subform, or Table for which to redraw the subform, or Field for which to redraw the area, or Variable for which to redraw the area, or Form table to redraw on a Web browser

Description

When you use a method to change the value of a field or subfield displayed in a subform, you must execute REDRAW to ensure that the form is updated.

Web Server: When executed after the On Timer form event, the REDRAW command can be called to periodically update a 4D form sent to a Web browser. For more information, please refer to the description of the SET TIMER command.

See Also

SET TIMER.

SCROLL LINES ({*; }object{; position{; *}})

Parameter	Type		Description
*	*	→	If specified, object is an object name (string) If omitted, object is a table or a variable
object	Form object	→	Object name (if * is specified) or Table or variable (if * is omitted)
position	Longint	→	Line position in the selection
*	*	→	Display the line in first position after scroll

Description

The SCROLL LINES command allows scrolling lines of a list form (displayed using the MODIFY SELECTION or DISPLAY SELECTION commands), a subform, a hierarchical list or a list box so that the first selected record/row or a specific record/row is displayed.

If you pass the first optional * parameter, you indicate that the object parameter is the name of a subform, a hierarchical list or list box object (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) or a variable (ListRef of a hierarchical list or list box).

If you do not pass the position parameter, the command provokes the vertical scroll of lines of the list so that the first highlighted line in the list is visible. If no line is selected, the command does nothing. If at least one selected line is already visible, the command does nothing.
The position parameter allows you to indicate the number of the line to display. 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. For list forms and subforms, this number is the number of the line among the current selection (its position). In the case of hierachical 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 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.

Note: The HIGHLIGHT RECORDS command features an optional * parameter that allows delegating scroll management to the SCROLL LINES command.

See Also

HIGHLIGHT RECORDS, SELECT LISTBOX ROW.

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 Developer 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 Developer, 4D Desktop, 4D Client and 4D Server. A new process is created when it is run on a server machine.

Examples

1. The following example replaces the About 4D Developer menu command with the About Scheduler menu command. The ABOUT method displays a custom About box:

```
SET ABOUT("About Scheduler…"; "ABOUT")
```

2. The following example resets the About 4D Developer menu command back to the original About box:

```
SET ABOUT("About 4D Developer";"")
```

SET CURSOR {(cursor)}

Parameter	Type		Description
cursor	Number	→	Mac OS-based cursor resource number

Description

The SET CURSOR command changes the mouse cursor to the cursor stored in the Mac OS-based 'CURS' resource whose ID number you pass in cursor.

If you omit the parameter, the mouse cursor is set to the standard arrow.

Use the RESOURCE LIST command to get the list of available cursors.

See Also

RESOURCE LIST.

SET FIELD TITLES (aTable | aSubtable; fieldTitles; fieldNumbers{; *})

Parameter	Type		Description
aTable | aSubtable	Table or Subtable	→	Table or Subtable for which to set the field titles
fieldTitles	String Array	→	Field names as they must appear in dialog boxes
fieldNumbers	Numeric Array	→	Actual field numbers
*		→	Use the custom names in the formula editor

Description

SET FIELD TITLES enables you to mask, rename, and reorder the fields of the table or subtable 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 via the 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 will 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 lets you indicate 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.

• If the * parameter is passed, the names defined by this command can be 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: At the formula editor level, the execution of this command without the * parameter does not modify any settings made previously with the * parameter. In other words, the formula editor always displays the custom name set via the last call of the command with the * parameter.

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 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 FIELD TITLES command is the worksession. One benefit in Client/Server is that several 4D Client 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.
• Each call to SET FIELD TITLES must be followed or preceded by a call to SET TABLE TITLES — even though you do not want to modify the table title — otherwise the command will have no effect.
• Plug-ins always access the "virtual" structure as specified by this command.

Example

See example for the SET TABLE TITLES command.

See Also

Field name, Get last field number, SET TABLE TITLES.

version 2004 (Modified)

Compatibility Note
This command is maintained only for compatibility reasons. In new databases (created with 4D starting with version 2004), the platform interface is managed automatically by the program and the command has no effect. It can still be used in converted database, although it is recommended to activate the new interface management feature in the Preferences dialog box ("System" option).

SET PLATFORM INTERFACE (interface)

Parameter	Type		Description
interface	Number	→	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

Description

The SET PLATFORM INTERFACE command sets the platform interface used for displaying the forms.

You can pass in interface one of the following predefined constants:

Constant	Type	Value
Automatic Platform	Long Integer	-1
Mac OS 7	Long Integer	0
Windows 3.11, NT 3.51	Long Integer	1
Windows 9x	Long Integer	2
Mac OS 9	Long Integer	3
Mac Theme	Long Integer	4

Note: The constant Mac Theme allows you to use the user interface defined with the *Appearance Manager*. This manager only exists on Mac OS. When a database defined with “Mac Theme” interface is displayed on Windows, the interface “Windows 9x” is applied.

The command does nothing if the value you pass does not change the current platform interface.

Note: The platform interface can also be changed in the **Preferences** dialog box.

Example

In a 4D Client/Server architecture, the Macintosh and Windows stations can use different platform interfaces concurrently. To do so, you can call the SET PLATFORM INTERFACE command in the On Startup Database Method:

```
    ` This example assumes that user preferences are stored in a [Preferences] table
    ` Look for the record corresponding to the current user
QUERY([Preferences];[Preferences]User name=Current User)
If (Records in selection([Preferences])=0)
        ` If not found, look for the default preferences
    QUERY([Preferences];[Preferences]User name="Default")
End if
    ` Set the Platform Interface according to the user preferences
SET PLATFORM INTERFACE ([Preferences]Platform Interface)
```

See Also

Get platform interface.

SET TABLE TITLES (tableTitles; tableNumbers{; *})

Parameter	Type		Description
tableTitles	String Array	→	Table names as they must appear in dialog boxes
tableNumbers	Numeric Array	→	Actual table numbers
*		→	Use the custom names in the formula editor

Description

SET TABLE TITLES enables you to mask, rename, and reorder the tables of your database when they appear in standard 4D dialog boxes such as the Query editor, within the Application environment (more specifically, when the editors are called via the 4D language commands). 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 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 will 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 lets you indicate 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.
• If the * parameter is passed, the names defined by this command can be 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: At the formula editor level, the execution of this command without the * parameter does not modify any settings made previously with the * parameter. In other words, the formula editor always displays the custom name set via the last call of the command with the * parameter.

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.

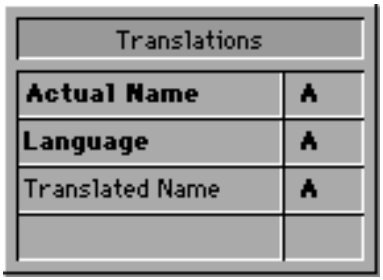
Use the SET TABLE TITLES command for:
• Dynamically localizing a database.
• Showing tables the way you want, independent from the actual definition of your database.
• Showing tables in a way that depends on the identity or custom privileges of a user.

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
          `desired language
      QUERY([Translations];[Translations]LanguageCode=$Language;*)
      QUERY([Translations]; & ;[Translations]TableID=$vlTable;*) `table number
         `field number = 0 means that it is a table name
      QUERY([Translations]; & ;[Translations]FieldID=0)
      If(Is table number valid($vlTable)) `check that the table still exists
         If(Records in selection([Translations])=0)
               ` Otherwise, create the record
            CREATE RECORD([Translations])
            [Traductions]LanguageCode:=$Language
            [Traductions]TableID:=$vlTable
            [Traductions]FieldID:=0
               ` The name of the translated table will need to be entered
            [Translations]Translation:=Table name($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
                         `Desired language
                    QUERY([Translations];[Translations]LanguageCode=$Language;*)
                    QUERY([Translations]; & ;[Translations]TableID=$vlTable;*)  `table number
                    QUERY([Translations]; & ;[Translations]FieldID=$vlField)  `field number
                    If(Is field number valid($vlTable;$vlField))
                        If(Records in selection([Translations])=0)
                                ` Otherwise, create the record
                            CREATE RECORD([Translations])
                            [Traductions]LanguageCode:=$Language
                            [Traductions]TableID:=$vlTable
                            [Traductions]FieldID:=$vlField
                                ` The name of the translated field will need to be entered
                            [Traductions]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;*)
         `avoid the zero that serves as table name
      QUERY([Translations]; & ;[Translations]FieldID#0)
      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.

See Also

Get last table number, SET FIELD TITLES, Table name.

Shift down → Boolean

Parameter	Type		Description
This command does not require any parameters			
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
```

See Also

Caps lock down, Macintosh command down, Macintosh control down, Macintosh option down, Windows Alt down, Windows Ctrl down.

SHOW MENU BAR

Parameter	Type	Description

This command 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 command HIDE MENU BAR.

See Also

HIDE MENU BAR, HIDE TOOL BAR, SHOW TOOL BAR.

SHOW TOOL BAR

Parameter	Type	Description

This command does not require any parameters

Description

The SHOW TOOL BAR command makes the toolbar visible.

If the toolbar was already visible, SHOW TOOL BAR does nothing.

See Also

HIDE MENU BAR, HIDE TOOL BAR, SHOW MENU BAR.

Tool bar height

Tool bar height → Longint

Parameter	Type		Description
This command does not require any parameters			
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 4D tool bar, expressed in pixels. If the tool bar is hidden, the command returns 0.

See Also

HIDE TOOL BAR, Menu bar height, SHOW TOOL BAR.

Windows Alt down → Boolean

Parameter	Type		Description
This command does not require any parameters			
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 command Shift down.

See Also

Caps lock down, Macintosh command down, Macintosh control down, Macintosh option down, Shift down, Windows Ctrl down.

Windows Ctrl down

Windows Ctrl down → Boolean

Parameter	Type		Description
This command does not require any parameters			
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 command Shift down.

See Also

Caps lock down, Macintosh command down, Macintosh option down, Shift down, Windows Alt down, Windows Ctrl down.

60

Users and Groups

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 adds the user accounts present in the BLOB users in the database. 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.

The following rules apply when adding a user account:
• The user ID is used as a reference. Users are processed in order according to their user ID.
• If the number already exists in the structure file, account information is updated if necessary, according to the information contained in the BLOB.
• If the number does not exist in the structure file, the user is created according to the information contained in the BLOB.
• If the number matches a user account that was deleted in the structure file, the account is updated according to the information contained in the BLOB.
• If the information contained in the BLOB indicates that the user account is deleted, the account is deleted from the structure file.
• Updated users are linked to groups according to the information in the BLOB.
• If a group does not exist, it is created.

If the command is executed correctly, the system variable *OK* is set to 1. Otherwise, it is set to 0.

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 using the following sequence:

```
DOCUMENT TO BLOB(mydoc; blob)
BLOB TO USERS(blob)
```

However, user and group files generated in 4D starting with version 2004 cannot be opened with a previous version.

See Also

USERS TO BLOB.

System Variables or 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{(user; password)}

Parameter	Type		Description
user	String | Num	→	Name or unique user ID
password	String	→	Password (not encrypted)

Description

CHANGE CURRENT USER is used to change the identity of the current user in the database, without having 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 is displayed. 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 **Application/Access** page of the database Preferences.

You can also pass the two optional user and password parameters in order 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 (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 (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.
It works as follows:
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:
- It is possible to 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 in order to check the validity of the entered characters (minimum number of characters, uniqueness, etc.).
- In order for the characters of passwords being entered to be masked on screen, you can use the FONT 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
```

See Also

CHANGE PASSWORD.

CHANGE LICENSES

Parameter	Type	Description

This command does not require any parameters

Description

The CHANGE LICENSES command displays the 4D Update License dialog box, which enables the user to activate plug-ins, the Web server or, with 4D Server, to add expansion numbers in order to increase the number of clients who can use the database and its plug-ins simultaneously.

Note: In 4D Developer and 4D Server, you can display this dialog box by selecting the **Update License...** command in the **Help** menu.

Using the CHANGE LICENSES command, you can display the 4D License dialog box from the Application environment.

CHANGE LICENSES is a convenient way to allow licensing in a compiled 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, please refer to the 4D Installation Guide.

Example

In a custom configuration or preferences dialog box, you include a button whose method is:

```
` bLicense button object method
CHANGE LICENSES
```

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
```

See Also

CHANGE CURRENT USER.

Current user

Current user → String

Parameter	Type		Description
This command does not require any parameters			
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 command User in group.

See Also

CHANGE CURRENT USER, CHANGE PASSWORD, User in group.

DELETE USER (userID)

Parameter	Type		Description
userID	Number	→	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 command GET USER LIST.

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.

Deleted user names no longer appear in the Password window displayed when the database is open or when you call CHANGE ACCESS. However, in order to maintain unique user ID numbers, the user account is kept in the password system. Deleted user names are displayed in green in the Users editor of the Design environment.

See Also

GET USER LIST, GET USER PROPERTIES, Is user deleted, Set user properties.

Error Handling

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

Parameter	Type	Description
This command does not require any parameters		

Description

EDIT ACCESS allows the user to edit the password system. When this command is executed, the Toolbox window with only the Users and User groups pages is displayed.

Note: This command opens a modal window. Consequently, you should not call it from another modal window; otherwise it will do nothing.

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
```

See Also

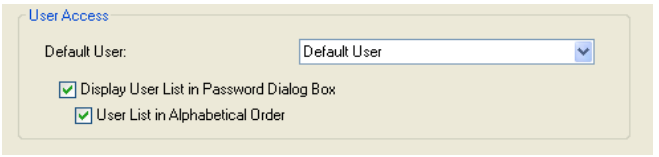
CHANGE CURRENT USER, CHANGE PASSWORD.

Get default user → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	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 (groupNames; groupNumbers)

Parameter	Type		Description
groupNames	String Array	←	Names of the groups as they appear in the Password editor window
groupNumbers	Numeric 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 (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 (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

See Also

GET GROUP PROPERTIES, GET USER LIST, Set group properties.

Error Handling

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 (groupID; name; owner{; members})

Parameter	Type		Description
groupID	Number	→	Unique group ID number
name	String	←	Name of the group
owner	Number	←	User ID number of group owner
members	Numeric 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 (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 (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 (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 Designer, user #-12 is the second, and so on).
15001 to 32767	Group created by the Designer or affiliated Group Owner (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 (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

See Also

GET GROUP LIST, GET USER LIST, Set group properties.

Error Handling

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 (plugIn) → String

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 Draw License	Longint	808464694
4D For OCI License	Longint	808465208
4D View License	Longint	808465207
4D Write License	Longint	808464697
4D Client Web License	Longint	808465209
4D Client SOAP License	Longint	808465465
4D ODBC Pro License	Longint	808464946
4D for ADO License	Longint	808465714
4D for MySQL License	Longint	808465712
4D for PostgreSQL License	Longint	808465713
4D for Sybase License	Longint	808465715

See Also

PLUGIN LIST, SET PLUGIN ACCESS.

GET USER LIST (userNames; userNumbers)

Parameter	Type		Description
userNames	String Array	←	User names as they appear in the Password editor window
userNumbers	Numeric 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 command Is user deleted 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 (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 Designer, user #-12 is the second, and so on).

See Also

GET GROUP LIST, GET USER PROPERTIES, Set user properties.

Error Handling

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 (userID; name; startup; password; nbLogin; lastLogin{; memberships{; groupOwner}})

Parameter	Type		Description
userID	Number	→	Unique user ID number
name	String	←	Name of the user
startup	String	←	Startup method name
password	String	←	Always an empty string
nbLogin	Number	←	Number of logins to the database
lastLogin	Date	←	Date of last login to the database
memberships	Numeric Array	←	ID numbers of groups to which the user belongs
groupOwner	Number	←	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 command GET USER LIST.

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 (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 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: GET USER PROPERTIES no longer returns the encrypted password in the password parameter. Starting with version 6.0.2, an empty string is always returned in this parameter. To check the password of a user, call 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 (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 (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

See Also

GET GROUP LIST, GET USER LIST, Set user properties, Validate password.

Error Handling

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 {(license)} → Boolean

Parameter	Type		Description
license	Number	→	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 enables you to know 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 may 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 Draw License	Longint	808464694
4D For OCI License	Longint	808465208
4D View License	Longint	808465207
4D Web License	Longint	808464945
4D Write License	Longint	808464697
4D Client Web License	Longint	808465209
4D Client SOAP License	Longint	808465465
4D SOAP License	Longint	808465464
4D ODBC Pro License	Longint	808464946
4D for ADO License	Longint	808465714
4D for MySQL License	Longint	808465712
4D for PostgreSQL License	Longint	808465713
4D for Sybase License	Longint	808465715

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.

See Also

Get plugin access, PLUGIN LIST, SET PLUGIN ACCESS.

Is user deleted (userNumber) → Boolean

Parameter	Type		Description
userNumber	Number	→	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.

See Also

DELETE USER, GET USER PROPERTIES, Set user properties.

Error Handling

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}) → Number

Parameter	Type		Description
groupID	Number	→	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	Number	→	User ID number of new group owner
members	Numeric Array	→	New group members
Function result	Number	←	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 (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 (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 (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 Designer, user #-12 is the second, and so on).
15001 to 32767	Group created by the Designer or affiliated Group Owner (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 (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.

See Also

GET GROUP LIST, GET GROUP PROPERTIES, GET USER LIST.

Error Handling

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 (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 lets you set, by programming, the user group allowed to use each “serialized” plug-in that is installed in the database. By doing so, you can manage how plug-in licenses are used.

Note: This operation can also be carried out 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 Draw License	Longint	808464694
4D For OCI License	Longint	808465208
4D View License	Longint	808465207
4D Write License	Longint	808464697
4D Client Web License	Longint	808465209
4D Client SOAP License	Longint	808465465
4D ODBC Pro License	Longint	808464946
4D for ADO License	Longint	808465714
4D for MySQL License	Longint	808465712
4D for PostgreSQL License	Longint	808465713
4D for Sybase License	Longint	808465715

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.

See Also

Get plugin access, PLUGIN LIST.

Set user properties

Set user properties (userID; name; startup; password; nbLogin; lastLogin{; memberships{; groupOwner}}) → Number

Parameter	Type		Description
userID	Number	→	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	Number	→	New number of logins to the database
lastLogin	Date	→	New date of last login to the database
memberships	Numeric Array	→	ID numbers of groups to which the user belongs
groupOwner	Number	←	ID number of user group owner
Function result	Number	←	Unique ID number of new user

Description

Set user properties enables you to change and update the properties of an existing user account whose unique user ID number you pass in userID, or to 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 (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 (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.

See Also

DELETE USER, GET GROUP LIST, GET USER LIST, GET USER PROPERTIES, Is user deleted, Validate password.

Error Handling

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) → Boolean

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"))
   OUTPUT FORM([Invoices];"Executive Output")
   INPUT FORM([Invoices];"Executive Input")
Else
   OUTPUT FORM([Invoices];"Standard Output")
   INPUT FORM([Invoices];"Standard Input")
End if
MODIFY SELECTION([Invoices];*)
```

See Also

Current user.

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 Aministrator.

Only the database Administrator or the Designer can execute this command. If another user attempts to execute it, the command does nothing and a privilege error (-9949) is generated.

The generated BLOB is automatically encrypted 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 allows the storing of user accounts in a BLOB field and not just in a file.

This concept allows you to keep a backup of users in the database data and, as such, implements a backup mechanism as well as a system to automatically load users in case of a database structure file update (information related to user accounts are stored by 4D in the database structure file).

See Also

BLOB TO USERS.

Validate password

Validate password (userID; password) → Boolean

Parameter	Type		Description
userID	Number	→	Unique user ID
password	String	→	Unencrypted password
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 is passed in userID.

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

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
```

See Also

GET USER PROPERTIES, Set user properties.

61

Variables

CLEAR VARIABLE

version 3

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 must be a process or an interprocess variable.

Note: You do not need to clear process variables when a process ends; 4D clears them automatically.

Local variables, which are variables preceded by a dollar sign ($), cannot be cleared with CLEAR VARIABLE. They are cleared automatically when the method in which are located 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
```

See Also

Undefined.

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 several variables from the document specified by document. The document must have been created using the command SAVE VARIABLES.

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 or Sets

If the variables are loaded properly, the OK system variable is set to 1; otherwise it is set to 0.

See Also

BLOB TO DOCUMENT, BLOB TO VARIABLE, DOCUMENT TO BLOB, RECEIVE VARIABLE, VARIABLE TO BLOB.

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 several variables in the document whose name you pass in document.

The variables do not need to be of the same type, but have to be of type String, Text, Real, Integer, Long Integer, Date, Time, Boolean, or Picture.

If you supply 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 VARIABLE or RECEIVE PACKET 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 or Sets

If the variables are saved properly, the OK system variable is set to 1; otherwise it is set to 0.

See Also

BLOB TO DOCUMENT, BLOB TO VARIABLE, DOCUMENT TO BLOB, LOAD VARIABLES, System Variables, VARIABLE TO BLOB.

Undefined (variable) → Boolean

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.
```

See Also

CLEAR VARIABLE.

62

Web Server

4D Developer, 4D Server and 4D Client 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.

• Contextual and Non-contextual mode
The 4D Web server can operate in two distinct modes: *contextual mode* and *non-contextual mode*. You can use the 4D Web server in either of these modes and you can pass from one mode to the other on the fly in accordance with your needs.
- **contextual mode** (available only with the Web server of 4D Developer and 4D Server) consists of a unique and unequaled feature. In this mode, 4D manages Web browsers as standard database clients. Your database is published directly on the Web. You do not need to develop a database, a Web site and then a CGI interface between the two. Your database is your Web site. Any modification made to the structure or data of the database is immediately passed on to all the browsers connecting to it. 4D converts database menu bars, forms and methods into HTML on the fly: it is not necessary to know HTML to be able to publish a 4D database on the Web. 4D automatically maintains a data use context for each Web browser (selections, variables, etc.). Note that in return, Web navigation in contextual mode includes specific constraints. For more information, refer to the Using the Contextual Mode section.
- Used in **non-contextual mode** (standard mode), the 4D Web server is a completely standard HTTP server: Web pages are sent without it being necessary to maintain context. You can access the data of the 4D database and build "semi-dynamic" HTML pages on the fly that include both static data and data coming from the database, before sending them on to the Web browsers. You can also send static Web pages that do not require any processing by the Web server.

• 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.

• 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.

• CGI Support
The 4D Web Server can call CGIs in a very simple way, as well as be called by other HTTP servers through CGIs.

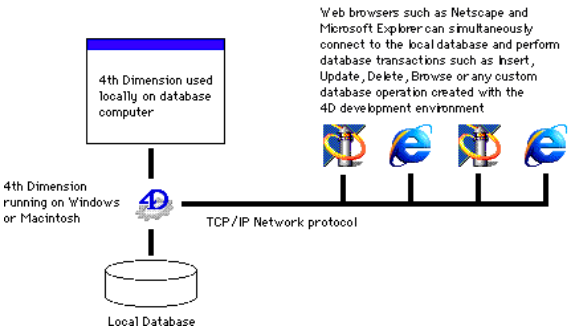
• Simultaneous operation of databases

4D Developre and the Web

If you publish a 4D database on the Web using 4D Developer, you can simultaneously:

• Use the database locally with 4D

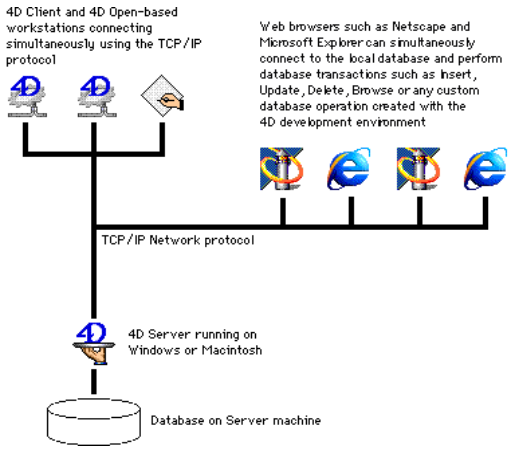
• Connect to the database using Web browsers.



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:

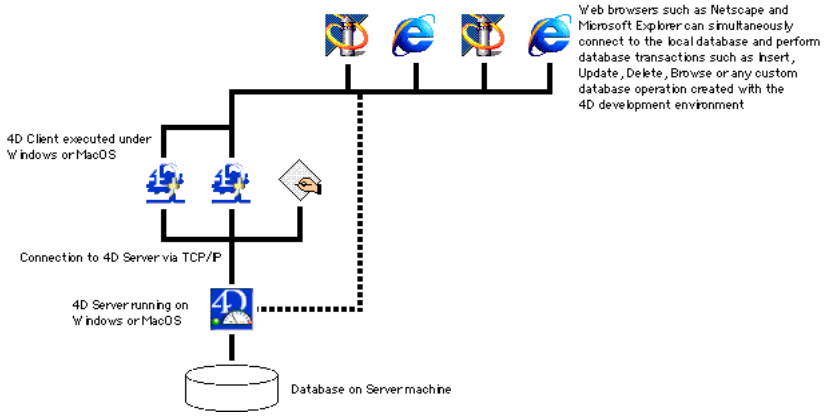
• 4D Client workstations
• 4D Open-based applications
• Web browsers.



4D Client and the Web

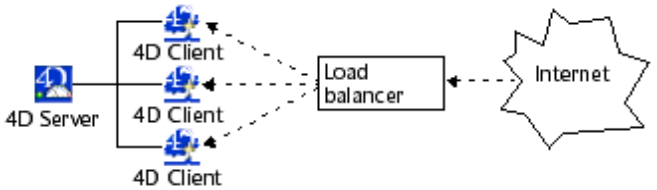
When a 4D database is published on the Web with 4D Client, it is possible to connect to the 4D database and to simultaneously use it:
- via 4D Client machines
- via applications using 4D Open

- 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 4D Client 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 Client, with the exception of the contextual mode. In fact, it is not possible to use the contextual mode with the 4D Client Web server (for more information about this mode, refer to the Using the Contextual Mode section).
Similarly, the operation of language commands is usually identical, whether the command be executed on 4D Developer, 4D Server or 4D Client. 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 Client**: since any 4D Client machine 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 Client 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 Client 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 protected 4D Client Web server can serve Intranet requests and another 4D Client Web server, located beyond the firewall, will serve Internet requests.
- the distribution of tasks between different 4D Client Web servers: one 4D Client Web server can be in charge of SOAP requests, another can handle standard requests, and so on.

See Also

Connection Security, SEND HTML FILE, SET HOME PAGE, SET HTML ROOT, SET WEB DISPLAY LIMITS, SET WEB TIMEOUT, STOP WEB SERVER, Using CGIs, Using SSL Protocol, Using the Contextual Mode, Web Server Settings.

version 2004 (Modified)

4D Developer, 4D Server and 4D Client include a Web server that enable you to publish the data of your databases on the Web, transparently and dynamically.
This section describes the steps necessary for publication of 4D databases and for connection of browsers, as well as the process of connection management.

Conditions for publishing a 4D database on the Web

To be able to publish a 4D database on the Web using 4D Developer, 4D Server or 4D Client, 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.

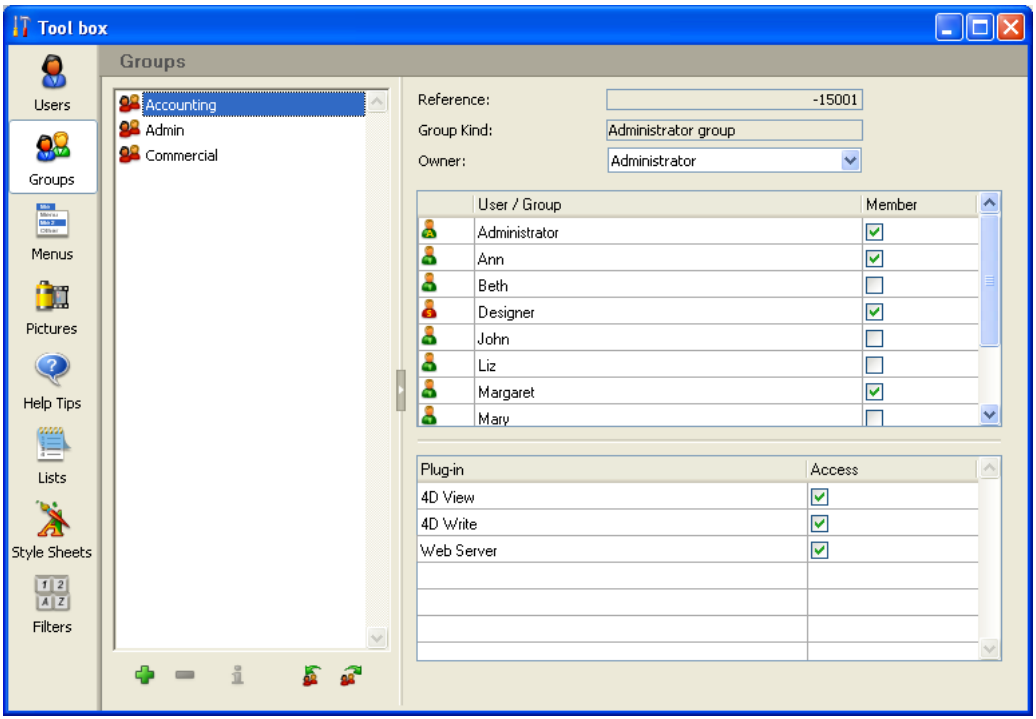
- If you want to use SSL for network connections, make sure that requested components are correctly installed (see section Using SSL Protocol).

• 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 Client)
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 4D Client by using the 4D password system.
In fact, 4D Client 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 Client (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 **Web Server** line in the Plug-in distribution area:



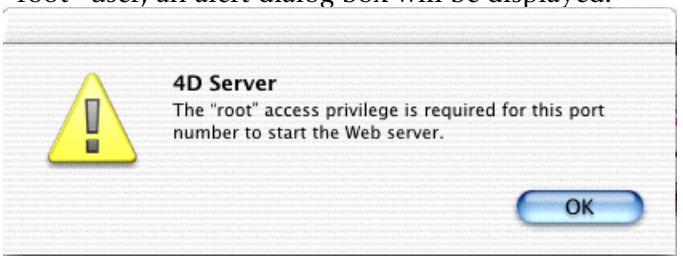
Above: only users belonging to the "Web" group are authorized to publish their 4D Client machine as a Web server.

Configuring the Web server under Mac OS X

Under Mac OS X, using TCP/IP ports reserved for Web publishing requires specific access privileges: only the “root” user of the machine can launch an application using these ports.

These ports are numbers 0 to 1023. Remember that, by default, a 4D database is published on TCP port 80 in standard mode and on port 443 in SSL mode.

Once you publish a 4D database on the default TCP port without being connected as the “root” user, an alert dialog box will be displayed:

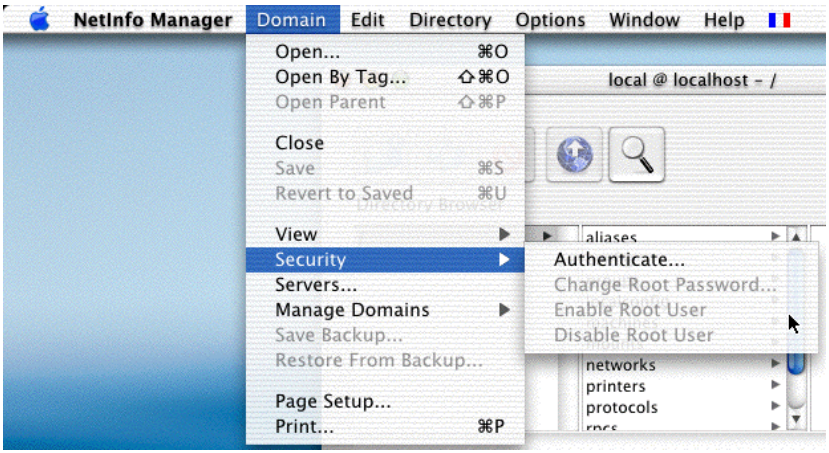


To use the Web server under Mac OS X, you have four possibilities:

• Modify the TCP port numbers used by the 4D Web server.
You must use port numbers greater than 1023, for example, port 8080 for standard mode and 8043 for SSL mode.
This operation occurs in the Preferences dialog box (see Web Server Settings section) or using the SET DATABASE PARAMETER command. In this case, it will be necessary to indicate the port number after each database connection URL (for example, http://www.mydatabase.com/pages/mypage.html:8080) and https://www.mydatabase.com/pages/payment.html:8043.

• Logging on as the “root” user
By default, the “root” user is not enabled on a machine running Mac OS X. You must first enable it and then log in with this user name.
Enabling a “root” user takes place using the **NetInfo Manager** utility provided by Apple and installed in the Applications:Utilities folder.

Once the utility has been launched, choose the **Security** command in the **Domain** menu, then the **Enable root user** option. You must have first identified the machine administrator using the **Authenticate...** command, located in the same menu (enter the shortened name and the administrator password).



For more information on this operation, refer to the Mac OS X documentation.

Once the “root” user has been created, you must close the session (Apple menu) and then log in using the “root” user name. You can then launch the Web server on port number 80, or a 4D Web server with a secure connection.

• **Port transfer**
This third solution lets you publish a 4D Web database under Mac OS X without being a “root” user and without it being necessary to specify the port number behind each connection URL to the server. It is based on port transfer. The principle consists of transferring, at the system level, the requests received on the standard TCP port number (80) to one specified in the 4D database (which must be greater than 1023). Note that this tip will not work with secured connections (the TCP port 443 is not modifiable).
To carry out this operation, you must connect as a “root” user, start the Terminal and use Unix commands.

To set up the port transfer under Mac OS X (assuming that your IP address is 192.168.93.45):
1. Open a session as a root user (see the previous paragraph).
2. Start the **Terminal** program.
This program is found in the Applications:Utilities folder.
3. Enter “su” (“substitute user” special account) followed by the root user password.

4. Enter the following command:
ipfw add 400 fwd 192.168.93.45,8080 tcp from any to 192.168.93.45 80
Of course, you must replace “192.168.93.45” with your own IP address.
The figure 400 is the reference number of this operation.
5. Quit the **Terminal** program.
6. Start your 4D application as a standard user.
7. In the Preferences of the database, set the Web publication TCP port to 8080.
From then on, Mac OSX is ready to transfer the requests received on port 80 to port 8080 instantaneously and in a manner which is invisible for the user.

To remove this mode of operation:
1. Start the **Terminal** program and enter:
ipfw delete 400
The requests received on port 80 will no longer be transferred to port 8080.

• Opening a temporary root session
This solution works as follows: the 4D Web server is initially launched in a “root” session that is opened for this purpose, but is closed soon after. This mechanism can be used with 4D Developer, 4D Server, 4D Client and the 4D Desktop executable applications.

Here are the details of the sequence:
1. The 4D application is executed using a classic user session.
2. When the Web server is launched on the standard port (port 80), an alert dialog box appears telling the user that the operation is not possible.
The dialog box gives the user the opportunity to modify access privileges in order to be able to launch the Web server. To do this, the user must enter an administrator name and password for the machine.
3. The user enters an administrator name and password for the machine.
Using this information, 4D can modify application access privileges and set the user session as “root”.
4. The user is asked to quit and restart the application.
5. On start-up, the 4D application starts in root session.
6. If the “Publish Database at Startup” option is checked in the application Preferences, the Web server is launched on port 80.
If the “Allow SSL for Web Server” option is checked, the SSL port (443 by default) is also opened.
7. After a few moments, the root session is automatically closed and replaced with the session of the current user.
The Web server remains published and the user session continues normally.

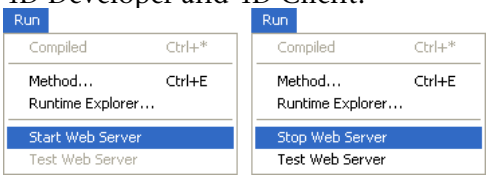
The drawback of this mechanism is that once it is in place, you cannot stop, start and restart the Web server at will during a session. You cannot go from a standard user session to a “root” session (higher access privileges) without restarting the application. This mechanism only works on application startup.

Starting the 4D Web Server

The 4D Web Server can be started in three different ways:

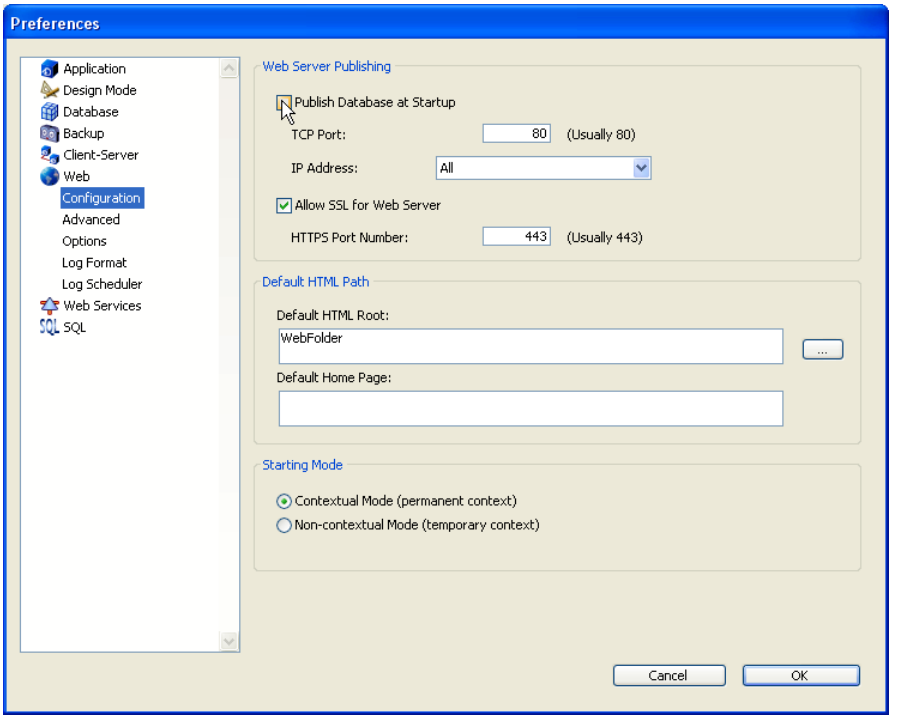
• Using the **Run** menu of 4D Developer and/or 4D Client or the **Web** menu of 4D Server. The Web Server menu allows you to start and stop the Web Server at your convenience:

4D Developer and 4D Client:



4D Server:

• Automatically publishing the database each time it is opened. To automatically publish a database on the Web, display the **Configuration** page of the **Web** theme of the application Preference of the application:



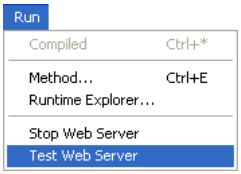
In the **Web Server Publishing** section, select the **Publish Database at Startup** check box, then click **OK**. Once this is done, the database will be automatically published on the Web each time you open it with 4D Developer, 4D Server or 4D Client.

• Programmatically, by calling the command START WEB SERVER.

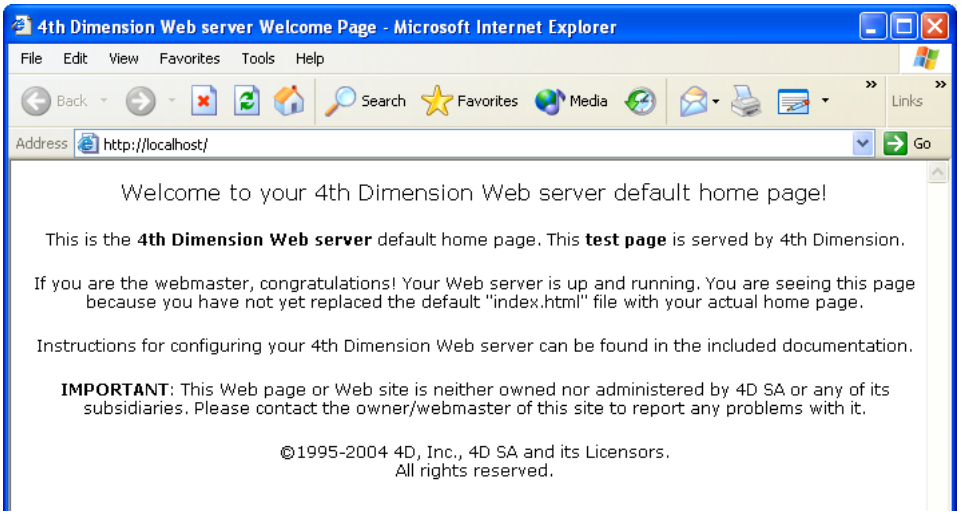
Tip: You do not need to quit 4D and reopen your database to start or stop publishing a database on the Web. You can interrupt and restart the Web server as many times as you want, using the **Run** menu or calling the commands START WEB SERVER and STOP WEB SERVER.

Testing the Web server

The **Test Web Server** command can be used to make sure the built-in Web server is functioning correctly (4D Developer and 4D Client 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 application preferences.

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 Developer, 4D Server or 4D Client 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 4).
• 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 Preferences (see Web Server Settings section) or using the SET HOME PAGE command.

3. You connect, but you get a Web page with the message “Menu Bar/This database is not ready to be published on the Web, you should first create a menu bar”. This means that you correctly connected to the database published in contextual mode but no home page nor menu bar has been defined (in contextual mode, 4D publishes menu bar #1 as the default home page if no HTML page is specified). For more information, see the Your First Time with the Web Server section.

4. 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 Web Server Settings section.

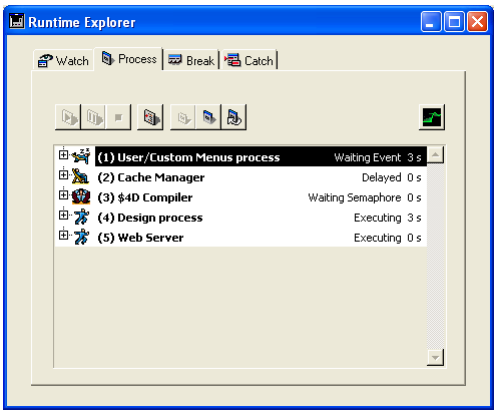
Note: If your database is protected by a password system, you may have to enter a valid user name and password (for more information, refer to section Connection Security).

Web Process management

Various 4D processes support Web publication of databases and connection to browsers. This paragraph describes these processes as well as their characteristics.

Web Server Process

The **Web Server** process runs and executes when the database is being published as a Web site. In the **Process** page of the Runtime Explorer window shown here, the Web Server process is the fifth process that is running and executing:



This is a 4D kernel process; you cannot abort this process using the **Abort** button. Also, you cannot attempt interprocess communication using commands such as CALL PROCESS. Note that the Web Server process does not have any user interface components (windows, menus, and so on).

You can start the Web Server process in the following ways:
• Choose **Start Web Server** in the **Web Server** menu of 4D Server or the **Run** menu of 4D Developer/4D Client.
• Call the 4D command START WEB SERVER.
• Open a database whose **Publish Database at Startup** Preference is checked.

You can stop running the Web Server process in the following ways:
• Choose **Stop Web Server** from the **Web Server** menu of 4D Server or the **Run** menu of 4D Developer/4D Client.
• Call the 4D command STOP WEB SERVER.
• Quit the database being currently published.

The purpose of the Web Server process is only to handle Web connection attempts. Starting the Web Server process does not mean that you open an actual Web connection, it just means that you allow Web users to initiate Web connections. Stopping the Web Server process does not mean that you close currently running Web connection processes (if any), it just means that you no longer allow Web users to initiate new Web connections.

If there are open Web connection processes when you stop the Web Server process, each of these processes continues executing normally.
Consequently, a delay time can be necessary to complete the termination of the Web Server process.

Web Connection Processes

Each time a Web browser attempts to connect to the database, the request is handled by the Web Server process, which performs the following steps:

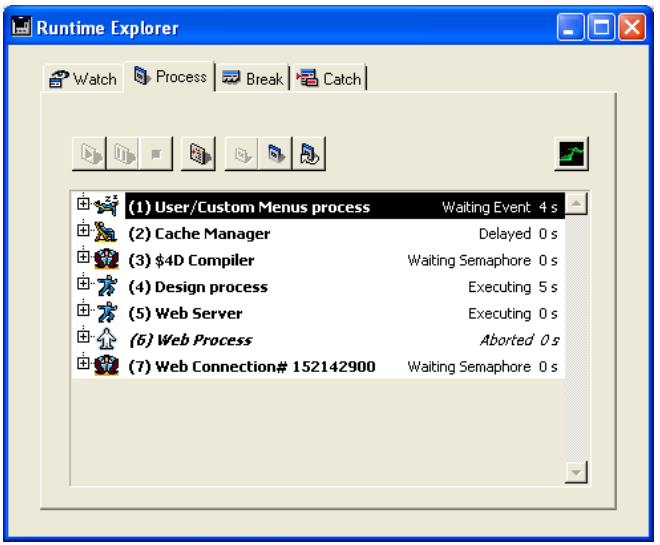
• First, it creates one or several temporary local 4D processes called **Web Processes** to evaluate and manage the connection with the Web browser.

Note: These temporary processes manage every HTTP request. They execute quickly and then aborted or delayed. For the Web server to be reactive in non-contextual mode, 4D freezes this "pool" of Web processes for 5 seconds and reuses them to execute any possible future HTTP queries. You can customize this behavior using the command SET DATABASE PARAMETER.

• If the request does not require that a context be created, 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).

• If the request requires that a context is created, it checks to verify that there are available resources for the new connection. If it is not the case, it sends the following message to the Web browser: “This database has not been setup for the Web yet.”

If the Web connection is initiated successfully, then a **Web Connection process** is started. This is the process that will handle the entire Web session for that connection. The Process list shown here displays the Web connection process “Web Connection# 152142900,” started after a Web browser connection has been initiated:



Note that the sixth process, which was started then aborted, handled the initialization of the Web connection.

Note: For more information about the context management, see the paragraph Using the Contextual Mode section.

• If during the session, the connection switches from contextual mode to non-contextual mode, the Web connection process (with an ID) is aborted.
Conversely, if during the session, the connection switches from non-contextual mode to contextual mode, a numbered Web connection process is created.

See Also

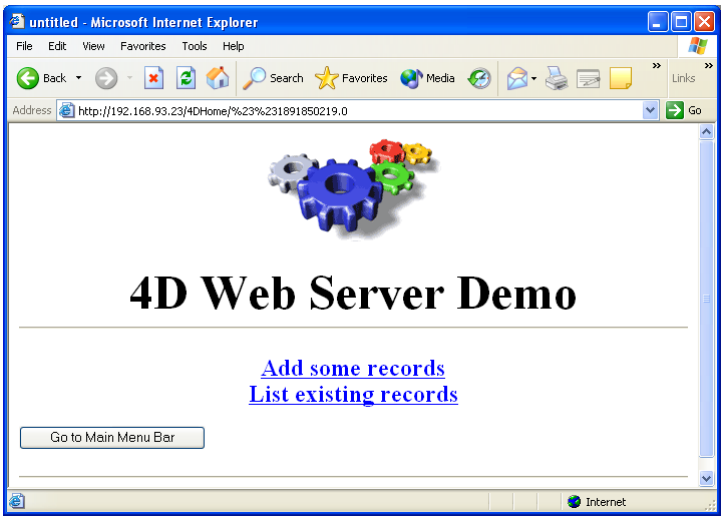
SEND HTML FILE, SET HTML ROOT, SET WEB DISPLAY LIMITS, SET WEB TIMEOUT, STOP WEB SERVER, Using SSL Protocol.

Example in contextual mode

This section gives a simple example of instant database publication in contextual mode. This automatic mode can be used more particularly for Intranet servers. It illustrates the basic principles of 4D Web server operation.

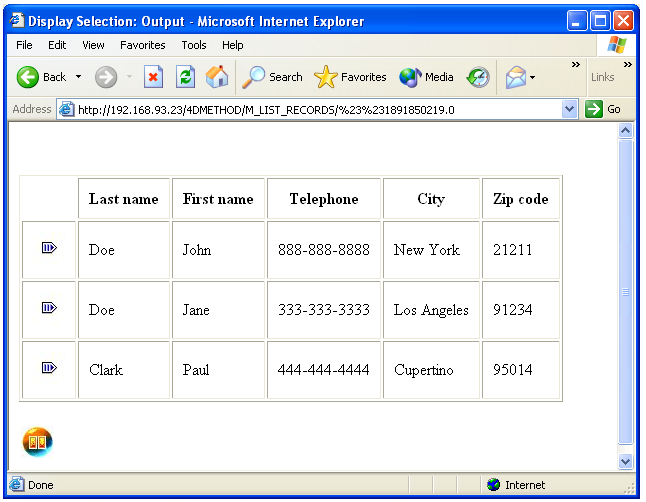
The structure of this database (given at the end of this section) is simple: the database is made of one table, an input form, an output form and a menu bar. The Home page is customized. The database is published as a Web server.

1. Connect to the Web server database
Connect to the Web Server by opening the database on a Web browser. You get the following Home Web page:



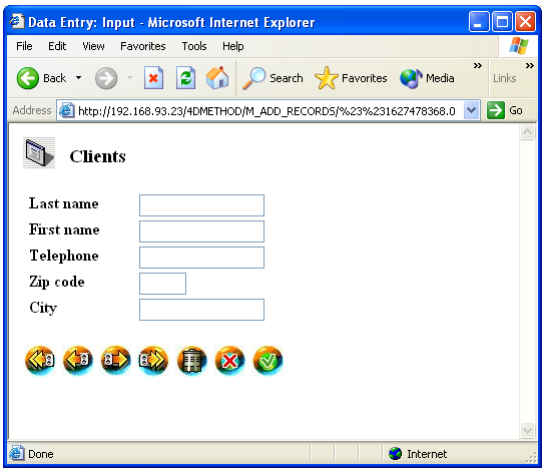
2. Display and browse the records

Click on the linked text List Existing Records. This presents the Web equivalent of a 4D Display Selection screen:



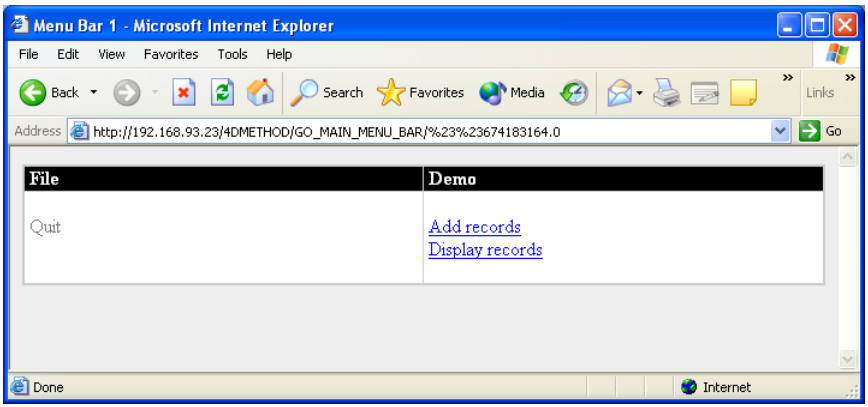
At this point, you can browse the records at your convenience. After you click the Done button, you go back to the Web site Home page.

3. Add records
In the Web site Home page, click on the Linked text Add Some Records to display the Web equivalent of a 4D Add Record screen:



You can add as many as records as you wish. When you are done, click the Cancel button (the one with the red cross) to return to the Web site Home page.

4. List or add records in the Main menu
In the Home page, click the Go to Main Menu Bar button. This exits the Home page and presents the Web equivalent of the 4D Custom menu bar:



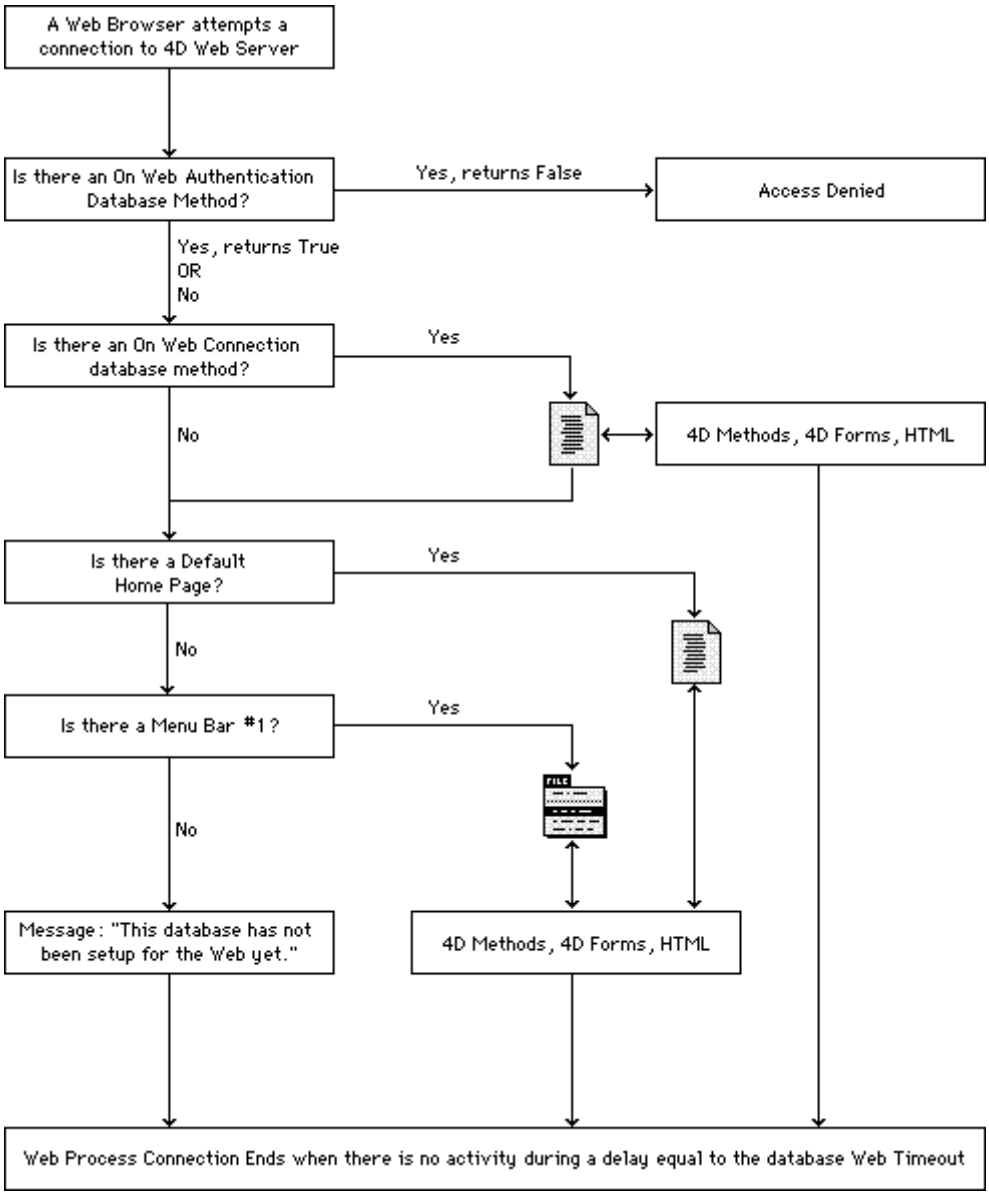
At this point, clicking on each menu item allows you to List or Add records: the same 4D methods that were used from the Home page are associated to the menu items.

5. Terminate the connection
When you are done, just quit your browser. 4D will terminate the Web connection process once the timeout delay has elapsed.

Initiating a Web connection in contextual mode

Each time a Web browser connects to a 4D database published as a Web Server in contextual mode, 4D performs the following actions:
• It executes the On Web Authentication database method, if it exists.
• If this method returns True or does not exist, it executes the On Web Connection database method, if it exists.
• If there is no such database method or if the method has completed, 4D then displays the Default Home page defined in the Preferences, if any.
• If no Default home page is defined, 4D then displays the current menu bar (by default menu bar #1), if it exists.
• If there is neither a Default Home page nor a menu bar, 4D displays a default Web page that states: *"This database has not been setup for the Web yet"*.

The following diagram summarizes these actions:



The On Web connection database method can call any of the project methods or forms defined in the database as well as HTML pages. The database method can actually handle the whole session.

A Web connection to 4D or 4D Server is not the same as a Client/Server connection. The HTTP protocol, which supports HTML and the Web, is not a "session-based" protocol; it is rather a "request-based" protocol. In Client/Server, you connect, work in a session, and then disconnect from the server. With HTTP, each time you perform an action that requires the attention of or an action from the Web Server, a request is sent to the server. In short, an HTTP request can be understood as the sequence "Connect+Request+Wait for reply+Disconnect."

In contextual mode, in order to run a Client/Server session via HTTP, by default 4D maintains, through a transparent encoding of the URLs, a context that uniquely identifies your Web connection and at the same time associates the connection to the 4D process handling the connection.

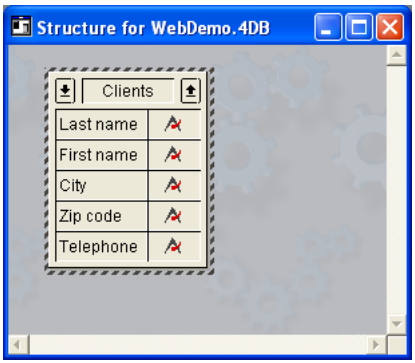
However, in this mode 4D has no way to provide an equivalent of the Client/Server disconnect action that terminates a session. That is the reason why the termination of a Client/Server session is handled through a timeout scheme. The 4D process handling the Web connection terminates after no activity has been detected for a delay time equal to the database Web timeout settings.

Database and Web Server in one

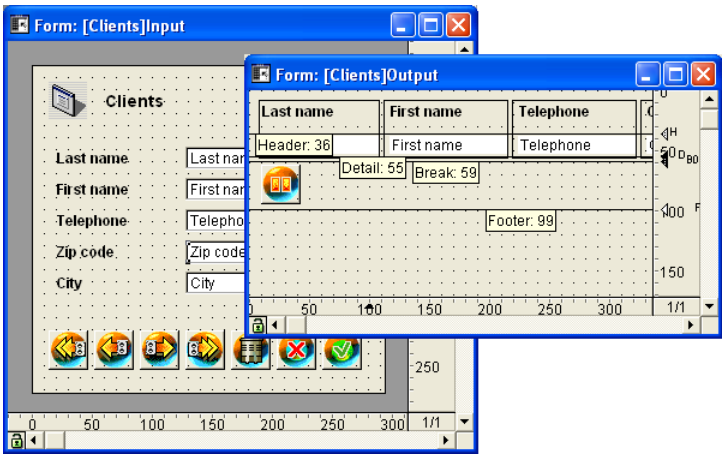
You can completely manage a 4D Web Server session using 4D Menus Bars, Forms and Methods in contextual mode. In the preceding example, listing and adding records was performed by simple 4D methods and forms. If we had not included an HTML home page, a Web browser would have obtained, upon connection, the menu bar #1 shown.

If we eliminate the HTML home page, building a Web Server supporting database Client/Server transactions consists of building a 4D database on Windows or Macintosh, for one or multiple users. The following steps explain the process of creating the example database in this way.

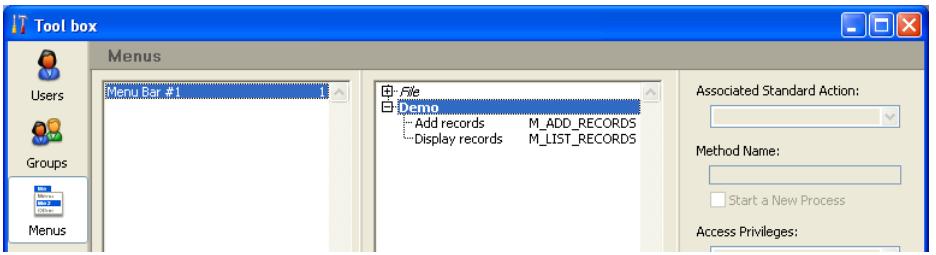
• Here is the Structure of the example database:



• Input and Output forms are added to enable you to work with records.



• Menu bar #1 is added to enable you to work with Custom menus and to support Web connections.

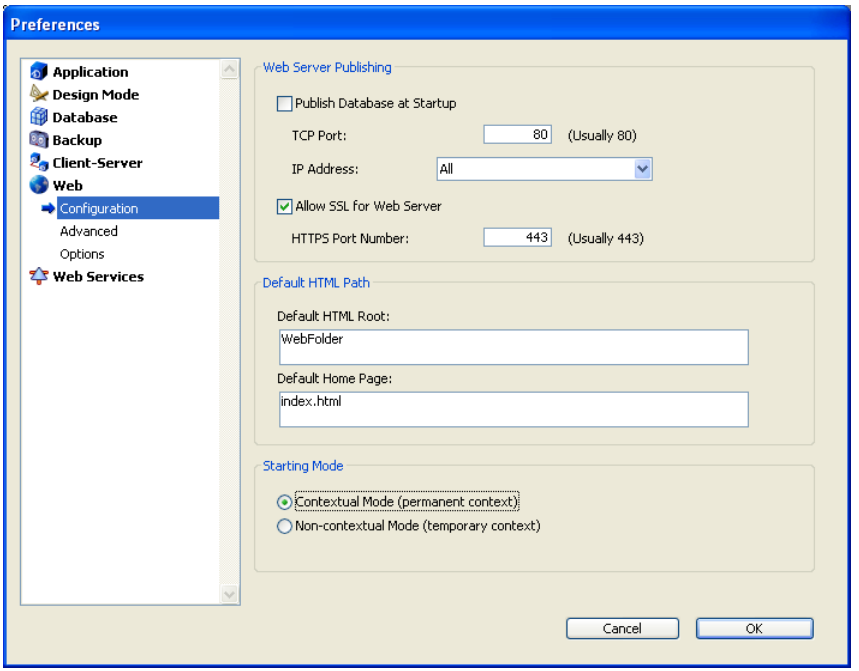


• The two following project methods are written.

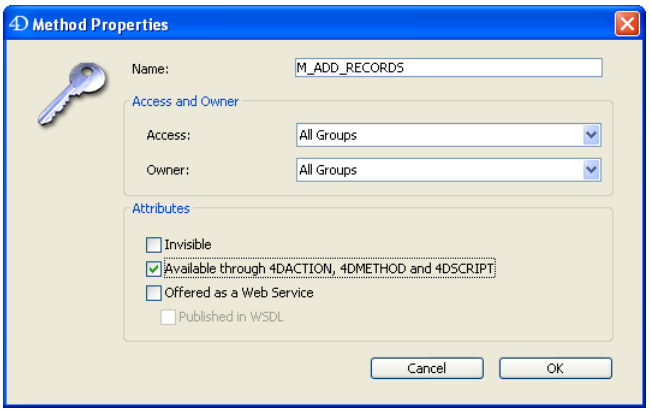
```
      ` M_ADD_RECORDS project method
C_TEXT($1)  ` This parameter MUST be explicitly declared
Repeat
   ADD RECORD ([Clients])
Until (OK=0)

      ` M_LIST_RECORDS project method
C_TEXT($1)  ` This parameter MUST be explicitly declared
ALL RECORDS ([Clients])
MODIFY SELECTION([Clients])
```

The "Start a New Process" attribute is assigned to each method.

• The Web server starts up in contextual mode and a default home page is defined in the database Preferences:



• The home page contains two links,“Add Some Records” and “List Existing Records,” that trigger the execution of the 4D project methods M_ADD_RECORDS and M_LIST_RECORDS through their URLs. The convention is quite simple: any HTML object can link to a project method of your database with the URL "/4DMETHOD/Name_of_your_Method". The **Available through 4DACTION, 4D METHOD and 4D SCRIPT** attribute must be associated with each method called using 4DMETHOD:



Once these links have been defined, when the Web browser sends back the URL, 4D executes the project method specified after the /4DMETHOD/ keyword. Then, after the project method has been completed, you go back to the HTML page that triggered its execution. Note that the project method can itself display 4D forms, other HTML pages, and so on.
Your 4D-based Web Site can be a completely 4D-based system or a combination of 4D forms and HTML pages. The interesting point in using HTML pages from within your 4D database is that you benefit from both the 4D and HTML development environments. Remember, you do not have to use HTML pages if you do not want to

The HTML home page in this example includes a button used to submit a record. There are three types of HTML buttons: **normal**, **submit**, and **reset**.

• Normal - Normal buttons can be attributed an URL that refers to a 4D method using the /4DMETHOD/ keyword. Normal buttons are used for navigation purposes.

• Submit - Submit buttons submit the form with the values entered by the user (if any) to the Web server. They are useful for handling data entry that you prefer to perform via an HTML page rather than a plain 4D form

• Reset - Reset buttons are not very useful within a 4D development: they clear the form of the values entered by the user (if any) and does not send any request to the server.

While integrating HTML pages into 4D, you will typically use normal or submit type buttons. The code of the home page button is the following: INPUT TYPE="SUBMIT" NAME="/4DMETHOD/GO_MAIN_MENUBAR"

To submit the HTML form on the 4D side, you need to specify the POST action 4D method that will be executed by 4D after the form is submitted.
To do this, it must contain the line FORM ACTION="/4DMETHOD/GO_MAIN_MENU_BAR" METHOD="POST"

- The GO_MAIN_MENU_BAR project method is the following:

```
SET HOME PAGE("")
```

In this example, this method has only one purpose: getting out of the current default home page displayed on the Web browser and then sending the current menu bar. 4D switches to the menu bar #1 of the database.

That is it!

In less than five minutes, you have created a 4D database that is both a locally operable database and a Web Server that you can publish on your Intranet network or on the Internet.

See Also

SEND HTML FILE, SET WEB DISPLAY LIMITS, SET WEB TIMEOUT, START WEB SERVER, STOP WEB SERVER.

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 4DACTION, 4DMETHOD and 4DSCRIPT" property for each project method of the database.

Note: The security of the connection itself can be managed through the SSL protocol. For more information, refer to section Using SSL Protocol.

Password Management System for Web Access

BASIC Mode and DIGEST Mode

In the database Preferences, 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 (starting with v11). The authentication mode concerns the way the information concerning the user name and password are collected and processed.

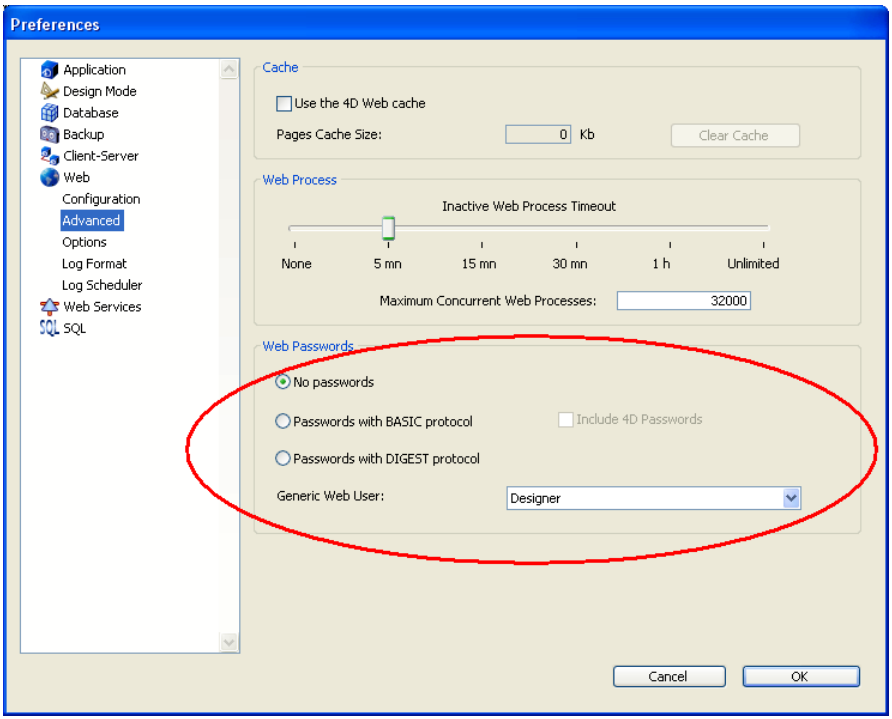
- In BASIC mode, the name and password entered by the user are sent unencrypted in the HTTP requests. This does not ensure total system security since this information could be intercepted and used by a third party.
- The DIGEST mode provides a greater level of security since the authentication information 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 Preferences dialog box, the access control system you want to apply to your Web server. To do this, in the Preferences dialog box, choose the **Advanced** 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.

Note: In this case, if the On Web Authentication Database Method doesn't exist, the connection is refused.

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 Validate Digest Web Password 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 Preferences 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.

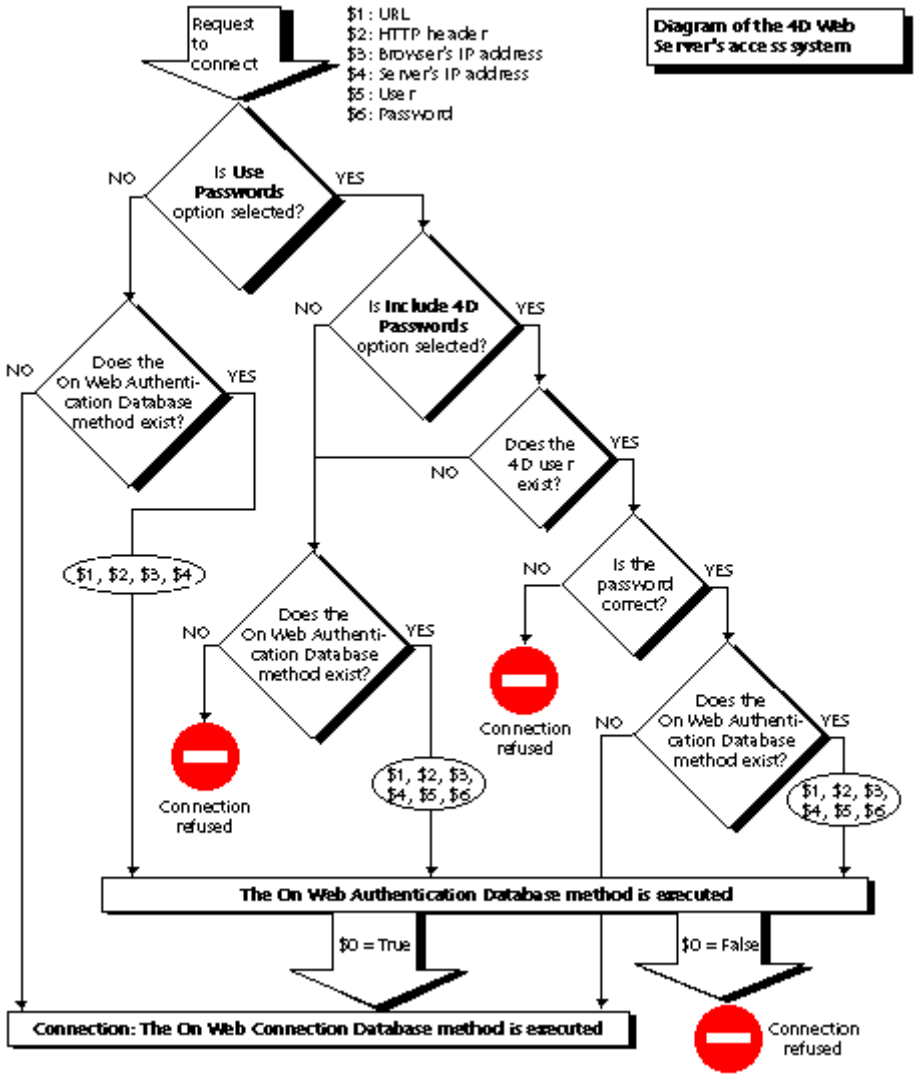
Note: If the user name sent by the browser is an empty string and if the On Web Authentication Database Method doesn't exist, a password dialog box is sent to the browser.

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 Validate Digest Web Password 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.

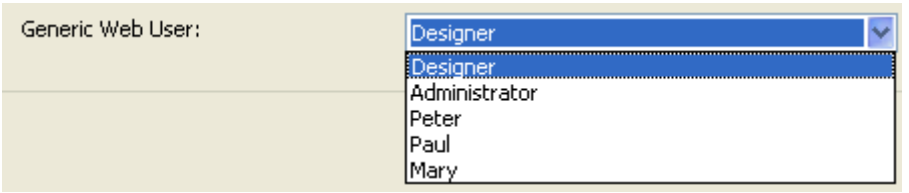
Generic Web User

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 database (tables, menus, 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 Preferences dialog, choose the **Advanced** 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 protocole BASIC

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.

Defining a HTML Root Folder by Default

This option in the Preferences 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 SEND HTML 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 Developer and 4D Server, at the same level as that of the database structure file.
• with 4D Client, at the same level as that of the 4D Client .exe file (under Windows) or the software package (under Mac OS).

You can modify the default HTML root folder name and location in the Preferences 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 Developer or 4D Server) or the folder containing the 4D Client application or software package (4D Client).
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 Client 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 Client 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 Client 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 Preferences dialog box, the cache is cleared so as to not store files whose access is restricted.

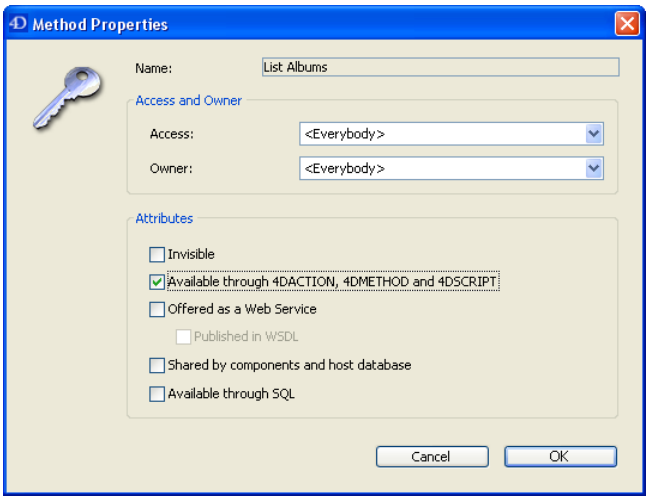
• You can also dynamically define the HTML root folder by using the SET HTML ROOT command. In this case, the modification applies to all the current Web process for the worksession. The cache of the HTML pages is therefore cleared.

Available through 4DACTION, 4DMETHOD and 4DSCRIPT

The special 4DACTION (non-contextual mode) and 4DMETHOD (contextual mode) URLs, as well as the 4DSCRIPT, 4DVAR and 4DHTMLVAR tags, allow you to trigger the execution of any project method of a4D 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 4DACTION, 4DMETHOD and 4DSCRIPT** 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 URLs, 4DACTION and 4DMETHOD, or the 4DSCRIPT, 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 or 4DMETHOD Web URLs or the 4DSCRIPT, 4DVAR and 4DHTMLVAR tags must be specifically indicated.

In the Explorer, Project methods “available through 4DACTION, 4DMETHOD and 4DSCRIPT” are given a specific icon:

See Also

On Web Authentication Database Method, On Web Connection Database Method, Using SSL Protocol.

On Web Authentication Database Method

Web Server

version 2004 (Modified)

The On Web Authentication Database Method is in charge of managing Web server engine access. It is called by 4D Developer, 4D Server or 4D Client when a Web browser request requires the execution of a 4D method on the server (method called using a 4DACTION or 4DCGI 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 (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
$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 will not eventually be filled in. The information received by the database method depends on the options that you have previously selected in the Preferences 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.

Note: 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.

• 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 Preferences 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.), otherwise it will be interrupted and the connection will be refused. The same is true if an error occurs while the database method is being executed.
• It is possible to forbid execution by 4DACTION or 4DMETHOD for each project method using the "Available through 4DACTION, 4DMETHOD and 4DSCRIPT" option in the Method properties dialog. For more information about this point, 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 4DMETHOD/
• when 4D receives a URL beginning with 4DCGI/
• when 4D receives a URL requesting a static page that does not exist
• 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.

Note that the On Web Authentication Database Method is NOT called when the server receives a URL requesting a valid static page.

Examples

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
```

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
```

See Also

Connection Security, Database Methods, On Web Connection Database Method, URLs and Form Actions.

The On Web Connection database method can be called in three different cases:
• the Web server receives a request beginning with the 4DCGI URL.
• the Web server receives an invalid request.
• it is also called by 4D Developer or 4D Server each time a Web browser initiates a connection to the database in contextual mode, or each time the Web server receives a request requiring the creation of a context (this case is not handled by 4D Client, which does not support the contextual mode).

For more information, refer to the paragraph "On Web Connection Database Method calls" below.

The request should have been previously accepted by the On Web Authentication Database Method (if it exists) and the database should be published as a Web server.

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 Netscape 4.5 running on Mac OS, you may receive a header similar to this:

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.5 (Macintosh; I; PPC)
Host: 123.45.67.89
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: us
Accept-Charset: iso-8
```

With Microsoft Internet Explorer 6 running on Windows, you may receive a header similar to this:

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Host: 123.45.67.89
Accept: image/gif, image/x-xbitmap, image/pjpeg, */*
Accept-Language: us
```

If your application deals with this information, it is up to you to parse the header and the body.

• 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.

• 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 Preferences 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. It also plays a role as the entry point in contextual mode (with 4D Developer and 4D Server).

Warning: Calling a 4D command that displays an interface element (ALERT, DIALOG...) ends the method processing.

The On Web connection database method is therefore called in the following cases:

• When connecting a browser to a 4D Web server operating in contextual mode. The database method is called with the /<action>... URL.

• When 4D receives the /4DMETHOD URL. The Web server switches to contextual mode and the database method is called with the /4DMETHOD/MethodName URL in $1.

• 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.

To know whether the On Web Connection Database Method was called from a contextual or from a non-contextual connection, you can use the Web Context function, that returns True if it is called from contextual mode, and False otherwise.

Consequently, we suggest that you structure the On Web Connection database method in the following manner:

```
    `On Web connection database method
C_TEXT($1;$2;$3;$4;$5;$6)
If (Web Context) `If in contextual mode
    WithContext ($1;$2;$3;$4;$5;$6)
        `The WithContext contains everything that was in the
        `On Web connection database method in 4D 6.0.x
Else
    NoContext ($1;$2;$3;$4;$5;$6)
        `The NoContext method executes the non-contextual
        `processing of requests (generally short)
End if
```

Example: Implementing Client Local Home Pages in contextual mode

In the following example, the parameter $1, sent to the On Web Connection database method, is used to implement Client Home Pages within an organization. The Intranet server operates in contextual mode.

The database has two tables: [Customers] and [Tables]. The On Startup database method shown here initializes interprocess arrays used later by the On Web Connection database method.

```
    ` On Startup Database Method

    ` Table List
ARRAY STRING(31;<>asTables;Get last table number)
For ($vlTable;1;Size of array(<>asTables);1;-1)
    If(Is table number valie($vlTable))
        <>asTables{$vlTable}:=Table name($vlTable)
    Else
        DELETE FROM ARRAY(<>asTables;$vlTable)
    End if
End for

    ` Standard Web Actions at Login
ARRAY STRING(31;<>asActions;2)
<>asActions{1}:="Add"
<>asActions{2}:="List"
```

The main job of the On Web Connection database method is to decipher the extra data passed in the URL after the host part of the address and to act accordingly. The method is as follows:

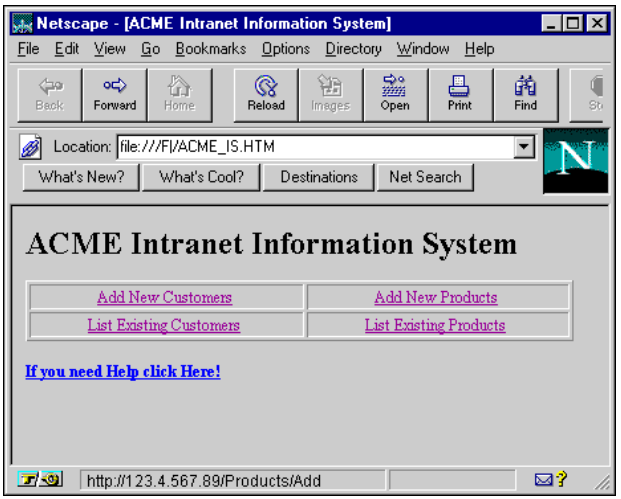
```
    ` On Web Connection Database Method

C_TEXT($1;$2;$3;$4;$5;$6)
C_TEXT($vtURL)
```

```
If (Web context) `If we are in contextual mode
        ` Just in case, check that $1 is equal to "/" or "/..."
    If ($1="/@")
            ` Copy the URL into a local variable minus the first "/"
        $vtURL:=Substring($1;2)
            ` Parse the URL and populate a local array with the tokens of the URL
            ` For example, if the URL extra data is "aaa/bbb/ccc", the resulting array
            ` will be of the three elements "aaa", "bbb" and "ccc" in that order
        $vlElem:=0
        ARRAY TEXT($atTokens;$vlElem)
        While ($vtURL # "")
            $vlElem:=$vlElem+1
            INSERT IN ARRAY($atTokens;$vlElem)
            $vlPos:=Position("/";$vtURL)
            If ($vlPos>0)
                $atTokens{$vlElem}:=Substring($vtURL;1;$vlPos-1)
                $vtURL:=Substring($vtURL;$vlPos+1)
            Else
                $atTokens{$vlElem}:=$vtURL
                $vtURL:=""
            End if
        End while
            ` If extra data was passed after the HOST part of the URL
        If ($vlElem>0)
                ` Using the interprocess array initialized in the On Startup DB method
                ` Check whether the 1st token is a name of a table
            $vlTableNumber:=Find in array(<>asTables;$atTokens{1})
            If ($vlTableNumber>0)
                    ` If so, get pointer to this table
                $vpTable:=Table($vlTableNumber)
                    ` Set the Input and Output forms
                INPUT FORM($vpTable->;"Input Web")
                OUTPUT FORM($vpTable->;"Output Web")
```

```
                    ` Using an interprocess array initialized in the On Startup DB Method
                    ` Check whether the 2nd token is a known standard action
                $vlAction:=Find in array(<>asActions;$atTokens{2})
                Case of
                        ` Adding records
                    : ($vlAction=1)
                        Repeat
                            ADD RECORD($vpTable->;*)
                        Until (OK=0)
                        ` Listing records
                    : ($vlAction=2)
                        READ ONLY($vpTable->)
                        ALL RECORDS($vpTable->)
                        DISPLAY SELECTION($vpTable->;*)
                        READ WRITE($vpTable->)
                    Else
                        ` Here could additional standard table actions be implemented
                End case
            Else
                ` Here could other standard actions be implemented
            End if
        End if
        ` Whatever happened above, pursue with the normal Log On process
    WWW NORMAL LOG ON
Else
    ... ` Here could the code managing the non-contextual mode
        `would be implemented
End if
```

At this point, people within the organization can connect to the database and enter a URL according to the convention set by the methods listed. Users can also create bookmarks if they do not want to re-enter the URL each time. In fact, the ultimate solution is to provide each member of the organization with an HTML page that they will use locally to access the database.

This HTML page is shown:



In other words, the HTML page ACME_IS.HTM is the Client Local Home Page for the 4D-based information system of the organization. If a user clicks on the Add New Products link, the Web browser will connect to the host having the URL http://123.4.567.89/Products/Add. Provided that the IP address of the database computer is 123.4.567.89, the On Web Connection Database Method receives the extra URL data "/Products/Add" in $1, and therefore proceeds to add records in the [Products] table.

Finally, users can drag and drop links from that page onto the desktop to create Internet Shortcut icons, such as the Add New Customers icon shown here. Simply double-clicking these icons will bring them directly into any part of your 4D Web database.

The source code of this HTML page is listed here:

```
<HTML>
<HEAD>
 <TITLE>ACME Intranet Information System</TITLE>
</HEAD>
<BODY>
<H1><B>ACME Intranet Information System</B></H1>
<P ALIGN=CENTER><TABLE BORDER=1 CELLPADDING=1 WIDTH="100%">
 <TR>
 <TD>
  <P ALIGN=CENTER><A HREF="http://123.4.567.89/Customers/Add">Add New Customers</A>
 </TD>
 <TD>
  <P ALIGN=CENTER><A HREF="http://123.4.567.89/Products/Add">Add New Products</A>
 </TD>
 </TR>
 <TR>
 <TD>
  <P ALIGN=CENTER><A HREF="http://123.4.567.89/Customers/List">List Existing Customers</A>
 </TD>
 <TD>
  <P ALIGN=CENTER><A HREF="http://123.4.567.89/Products/List">List Existing Products</A>
 </TD>
 </TR>
</TABLE></P>
<P><B><A HREF="Help.HTM">If you need Help click Here!</A></B></P>
</BODY>
</HTML>
```

See Also

Database Methods, On Web Authentication Database Method, URLs and Form Actions, Using the Contextual Mode.

Binding 4D objects with HTML objects

This section describes the means made available by the 4D Web server for exchanging information via the Web, i.e. for dynamically sending and receiving values. The following points will be dealt with:
• Sending dynamic values stored in 4D variables
• Receiving dynamic values via Web forms
• Using the COMPILER_WEB project method
• Managing server-side image mapping
• Embedding JavaScript

Note: the sending and receiving of dynamic values can be carried out automatically in contextual mode via converted 4D forms. For more information on this point, refer to the section Using the contextual mode.

Sending dynamic values

References to 4D variables can be inserted in your HTML pages. You can bind these references with any type of HTML object. When the Web pages are sent to the browser, 4D will replace these references with the current values of the variables. The pages received are therefore a combination of static elements and values coming from 4D. This type of page is called semi-dynamic.

Notes:
• You work with process variables.
• As HTML is a word processing oriented language, you will usually work with Text variables. However, you can also use BLOBs variables (which avoid the 32 000 characters limitation of text type variables). You just need to generate the BLOB in Text without length mode.

First, an HTML object can have its value initialized using the value of a 4D variable.

Second, after a Web form is submitted back, the value of an HTML object can be returned into a 4D variable. To do so, within the HTML source of the form, you create an HTML object whose name is the same as the name of the 4D process variable you want to bind. That point is studied further in the paragraph “Receiving dynamic values” in this document.

Note: In non-contextual mode, it is not possible to make a reference to 4D picture variables.

Since an HTML object value can be initialized with the value of a 4D variable, you can programmatically provide default values to HTML objects by including <!--#4DVAR 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 <!--#4DVAR VarName--> in the **value** field of HTML objects. In this case, you will have to type it in the HTML code.

The <!--#4DVAR --> 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. For more information, refer to the section 4D HTML Tags.

In fact, the syntax <!--#4DVAR VarName--> allows you to insert 4D data anywhere in the HTML page. For example, if you write:

<P>Welcome to <!--#4DVAR 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 send 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="/4DMETHOD/WWW_STD_FORM_POST" method="POST" name="frm">

<P><INPUT TYPE="hidden" NAME="vs4D" VALUE="<!--4DVAR 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>
```

In the HTML source code shown, note the **hidden** input object named vs4D. The value of this object is set to the text value "<!--#4DVAR vs4D-->". Since the project method sending the HTML file has previously defined the 4D process variable vs4D, 4D replaces the value of the HTML object and sets it to "4D4D", the value of the 4D variable.

The embedded JavaScript function Is4DWebServer tests the value of the vs4D HTML object. Here is the trick: if the HTML page is served by 4D, the object's value is changed to "4D4D". However, if the HTML page is served by another application (i.e., 4D WebSTAR on Macintosh), the object stays with its value as defined in the page, "[vs4D]". Bingo! By using JavaScript to test the value of that object, from within the page on the Web Browser side, you can detect whether or not the page is being served by 4D.

This first example shows how you can build "intelligent" HTML pages that provide additional features when being served by 4D, while staying compatible with other Web servers.

Important: You bind process variables only. In addition, the current version of 4D does not allow you to bind a 4D array to an HTML SELECT object. On the other hand, each element of a SELECT object can refer to separate 4D variables (i.e., the first element to V1, the second to V2, and so on).

The binding in the direction 4D toward Web Browser works with any encapsulation method (SEND HTML FILE, SEND HTML BLOB, as well as, in contextual mode, static text or text or BLOB variable in a 4D form).

Parsing of pages sent by the server

• In contextual mode, before sending an HTML page (HTML document or translated 4D form), 4D always parses the HTML source code in order to look for objects referring to 4D variables.
• In non-contextual mode, for the purpose of optimization, the parsing of the HTML source code is not carried out by the 4D Web server when HTML pages are called using simple URLs that are suffixed with ".HTML" or ".HTM". Of course, 4D offers mechanisms that allow you to "force" the parsing of pages when necessary (refer to the section 4D HTML Tags).

Inserting HTML Code into 4D Variables

You can insert HTML code into 4D variables. When the HTML static page is displayed on the Web browser, the value of the variable is replaced by the HTML code and will be interpeted by the browser.

To insert HTML code into 4D variables, you have two possibilites:
• Make the 4D variable start with code 1 as first character (i.e., vtHTML:=Char(1)+"...HTML code...") and add it to the HTML page using the <!--#4DVAR vtHTML--> tag.

• Insert directly the 4D variable (i.e., vtHTML:="...HTML code...") in the HTML page using the <!--#4DHTMLVAR vtHTML--> tag.
You can use a Text or a BLOB variable (the BLOB should have been generated in Text without length mode).

For more information, refer to section "HTML Tags".

Receiving dynamic values

When you send an HTML page using SEND HTML FILE or SEND HTML BLOB, you can also bind 4D variables with HTML objects in the "Web Browser toward 4D" direction. The binding works both ways: once the HTML form is submitted, 4D can copy back the values of the HTML objects into the 4D process variables. With a view to database compilation, these variables must be declared in the COMPILER_WEB method (see paragraph below).

It is also possible to retrieve values from the Web forms sent to 4D without prior knowledge of the fields that they contain, using the GET WEB FORM VARIABLES command. For more information, refer to the description of this command.

Warning: Getting the values back into the 4D process variables is only possible with HTML pages sent using SEND HTML FILE or SEND HTML BLOB. With HTML encapsulated in a 4D form in contextual mode, getting back values is restricted to the actual 4D objects located in the form.

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="/4DMETHOD/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 TYPE="text" NAME="vtUserName" VALUE="<!--4DVAR vtUserName-->" SIZE=30></P>
<P>
   <INPUT TYPE="submit" NAME="vsbLogOn" VALUE="Log On" onclick="return LogOn(frmWelcome)">
   <INPUT TYPE="submit" NAME="vsbRegister" VALUE="Register">
   <INPUT TYPE="submit" NAME="vsbInformation" VALUE="Information">
</P>

<P>
   <INPUT TYPE="hidden" NAME="vtNav_appName" VALUE="">
   <INPUT TYPE="hidden" NAME="vtNav_appVersion" VALUE="">
   <INPUT TYPE="hidden" NAME="vtNav_appCodeName" VALUE="">
   <INPUT TYPE="hidden" NAME="vtNav_userAgent" VALUE="">
</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.
• The initial value of the object vtUserName is <!--#4DVAR vtUserName-->.

Let's examine the 4D method WWW Welcome that sends this HTML page using the SEND HTML FILE command. This method is called by the On Web Connection Database Method.

```
` WWW Welcome Project Method
` WWW Welcome -> Boolean
` WWW Welcome -> Yes = Can start a session

C_BOOLEAN($0)
$0:=False
```

```
    ` Hidden INPUT HTML objects returning Browser information
C_TEXT(vtNav_appName;vtNav_appVersion;vtNav_appCodeName;vtNav_userAgent)
vtNav_appName:=""
vtNav_appVersion:=""
vtNav_appCodeName:=""
vtNav_userAgent:=""

    ` Text INPUT HTML object where the user name is entered
C_TEXT(vtUserName)
vtUserName:=""

    ` HTML submit button values
C_STRING(31;vsbLogOn;vsbRegister;vsbInformation)

Repeat
        ` Do not forget to reset the values of the submit buttons!
    vsbLogOn:=""
    vsbRegister:=""
    vsbInformation:=""
        ` Send the Web page
    SEND HTML FILE("Welcome.HTM")
        ` Test the values of the submit buttons in order to detect which one was clicked
    Case of

            ` The Log On button was clicked
        : (vsbLogOn # "")
            QUERY([WWW Users];[WWW Users]User Name=vtUserName)
            $0:=(Records in selection([WWW Users])>0)
            If ($0)
                WWW POST EVENT ("Log On";WWW Log information )
                    ` The method WWW POST EVENT saves information to a
                    ` table of the database
            Else
                CONFIRM("This User Name is unknown, would you like to register?")
                $0:=(OK=1)
                If ($0)
                    $0:=WWW Register
                        ` The method WWW Register allow a new Web User to register
                End if
            End if
```

```
            ` The Register button was clicked
        : (vsbRegister # "")
            $0:=WWW Register

                ` The Information button was clicked
        : (vsbInformation # "")
            DIALOG([User Interface];"WWW Information")
    End case
Until (Not(<>vbWebServicesOn) | $0)
```

The features of this method are:

• The 4D variables vtNav_appName, vtNav_appVersion, vtNav_appCodeName, and vtNav_userAgent (bound to the HTML objects having the same names) use the GetBrowserInformation JavaScript script to get back the values assigned to the HTML objects. Simple and direct, the method initializes the variables as strings, then gets back the values after the Web page has been submitted.

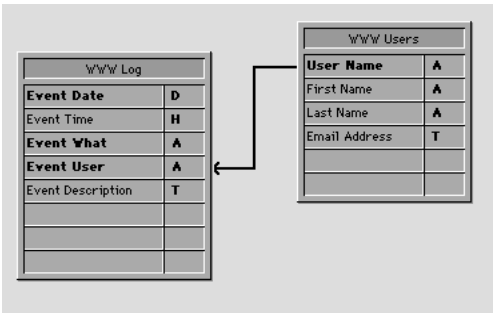
• The 4D variables vsbLogOn, vsbRegister and vsbInformation are bound to the three Submit buttons. Note that these variables are reset each time the page is sent to the browser. When the submit is performed by one of these buttons, the browser returns the value of the clicked button to 4D. The 4D variables are reset each time, so the variable that is no longer equal to the empty string tells you which button was clicked. The two other variables are empty strings, not because the browser returned empty strings, but because the browser "said" nothing about them; consequently, 4D left the variables unchanged. That is why it is necessary to reset those variables to the empty string each time the page is sent to the browser.

This the way to distinguish which Submit button was clicked when several Submit buttons exist on the Web page. Note that 4D buttons in a 4D form are numeric variables. However, with HTML, all objects are text objects.

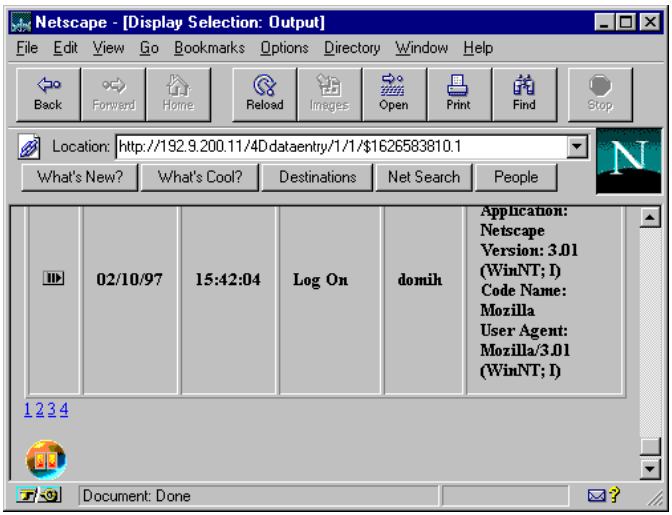
If you bind a 4D variable with a SELECT object, you also bind a text variable. In 4D, to test which element of a drop-down list was chosen, you test the numeric value of the 4D array. With HTML, this is the value of the selected item that is returned in the 4D variable bound to the HTML object.

No matter which object you bind with a 4D variable, the returned value is of type Text, so you bind String or Text 4D process variables.

An interesting point of this example is that after you have obtained information about the Browser, you can store these values in a 4D table, again combining Web and database capabilities. This is what the (unlisted) WWW POST EVENT project method does. It does not "post an event"; it saves the web session information into the tables shown here:



After you have saved the information in a table, you can use other project methods to send the information back to the Web user. To do so, simply use QUERY to find the right information and then use DISPLAY SELECTION to show the [WWW Log] records. The following figure shows the log information available to the registered user of the Web site:



Using the binding features shown in this example, combined with all the information you can give to or gather from users via HTML dialogs or 4D forms, you can add some very interesting administrative capabilities to your database Web site.

COMPILER_WEB Project Method

When the 4D Web server receives a posted form, it automatically calls the project method called **COMPILER_WEB** (if it exists). This method must contain all the typing and/or variable initialization directives, which are the variables whose names are the same as the field names in the form. It will be used by the compiler when compiling the database.
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.

Note: You can also use the GET WEB FORM VARIABLES command, which gets the value for all the variables included in a submitted HTML page.

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.

Binding HTML Objects with 4D Variables - Image Mapping

As seen in the section Using the Contextual Mode, when a 4D form is used as a Web page, 4D provides Server-side Image Mapping by means of invisible-like buttons that overlap a static picture.

If you send an HTML document using SEND HTML FILE or SEND HTML BLOB, you can bind 4D variables with Image Map HTML objects (INPUT TYPE="IMAGE") to retrieve information. For example, you can create an Image Map HTML object named bImageMap (you can actually use any name). Each time you click on the image on the browser side, a submit with the click position is sent back to the 4D Web Server. To retrieve the coordinates of the click (expressed relative to the top left corner of the image), you just need to read the value the 4D process variables bImageMap_X and bImageMap_Y (of type Longint) which contain the horizontal and vertical coordinates of the click. These variables should be declared in the *COMPILER_WEB* project method (see previous paragraph).

• In the HTML page, you write something like:

```
<P><INPUT TYPE="image" SRC="MyImage.GIF" NAME="bImageMap" BORDER=0></P>
```

• The 4D method that sends the HTML page contains:

```
SEND HTML FILE("ThisPage.HTM")
```

• In the *COMPILER_WEB* project method, you write:

```
C_LONGINT(bImageMap_X;bImageMap_Y)
bImageMap_X:=-1    `Initializing the variable
bImageMap_Y:=-1    `Initializing the variable
```

• Then, in the POST action 4D method or in the current method, after the POST action method issued a SEND HMTL FILE("") call, you retrieve the coordinates of the click in the bImageMap_X and bImageMap_Y variables :

```
If (($bImageMap_X#-1)&($bImageMap_Y#-1))
    ` Do something accordingly to the coordinates
End if
```

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 SEND HTML FILE or SEND HTML BLOB in standard mode, 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.

In contextual mode, if you encapsulate HTML in a 4D form, you do not have control over the HEAD section or the FORM declaration. The scope of the scripts is therefore different. For example, you cannot access the HTML form by its name. However, compare the Is4DWebServer JavaScript function of the previous example with this one:

```
function Is4DWebServer(){
   return (document.forms[0].vs4D.value=="4D4D")
}
```

Both functions do the same thing, but the second example uses the forms property of the HTML document object to access the object through the element forms[0]. As a result, it operates even if you do not know the name that 4D may or may have not given to the translated HTML page (form).

Note: 4D supports Java applets transport.

See Also

4D HTML Tags, SEND HTML BLOB, SEND HTML FILE, 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, in both contextual and non-contextual modes.

These URLs are the following:
• 4DMETHOD/, to link any HTML object to a project method of your database in contextual mode,
• 4DACTION/, to link any HTML object to a project method of your database in non-contextual mode,
• 4DCGI/, to call the On Web Connection Database Method from any HTML object.

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* manuel.

URL 4DACTION/

Syntax: 4DACTION/MyMethod{/Param}
Mode: Non-contextual. When called from contextual mode, aborts the context process and switches to non-contextual mode.
Usage: URL or Form action.

This URL allows you to link an HTML object (text, button...) to a 4D project method in non-contextual mode. 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 SEND HTML FILE or SEND HTML 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...).

Warning: For a 4D method to be able to be executed using the 4DACTION/ URL, it must have the "Available through 4DACTION, 4DMETHOD and 4DSCRIPT" attribute (unchecked by default), defined in the Method properties. For more information on this point, refer to the Connection Security section.

Example

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;$BlobVar)
SEND HTML BLOB ($BlobVar;"Pict/gif")
```

4DACTION to post forms

The 4D Web server offers an additional possibility when you want to use "posted" forms, which are static HTML pages that send data to the Web server. 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 **COMPILER_WEB** project method (if it exists, see below), then the On Web Authentication Database Method (if it exists). If it returns True, the MethodName method is executed. 4D analyzes the HTML fields present in the form, retrieves their values and automatically fills the 4D variables with their contents. The field in the form and the 4D variable must have the same name.

Note: For more information, refer to the section Binding 4D objects with HTML objects.

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">

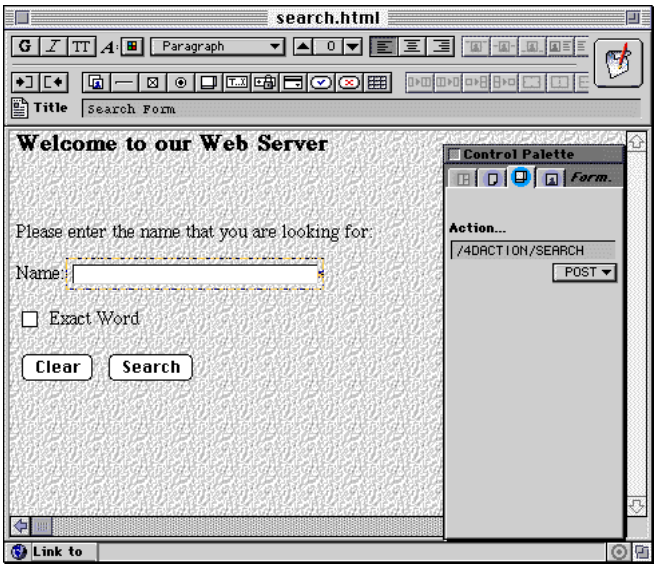
For each field in the form, 4D sets the value of the field to the value of the variable with the same name. For the form options (for example, check boxes), 4D sets the associated variable to 1 if it is selected, otherwise 0.
For numerical data entries, 4D converts the value of the field from Alpha–>Real.

Note: If a field in the form is named OK (for example the Submit button), the *OK* system variable is set to 1 if the value of the field is not empty, otherwise it is set to 0.

Example

In a 4D Web database started and used in “non-contextual” mode, we hope that the browsers can search 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 page as it appears in an HTML editor, in our case Adobe® PageMill™:



Here is the HTML code that corresponds to this page:

```
<FORM ACTION="/4DACTION/PROCESSFORM" METHOD=POST>
<INPUT TYPE=TEXT NAME=VNAME VALUE=""><BR>
<!-- Usually we put the name of the button in VALUE, for interpretation reasons, you must put a number in VALUE-->
<INPUT TYPE=CHECKBOX NAME=EXACT VALUE="1">Whole word<BR>
<!-- OK is a particular case-->
<INPUT TYPE=SUBMIT NAME=OK VALUE="Search">
</FORM>
```

During data entry, type “ABCD” in the data entry area, check the option and validate it by clicking the **Search** button.

4D then calls the *COMPILER_WEB* project method, which is as follows:

```
C_TEXT(VNAME)
VNAME:=""
C_LONGINT(vEXACT)
vEXACT:=0
OK:=0   `particular case
```

In the example, *VNAME* contains the string “ABCD”, *vEXACT* is equal to 1 and *OK* is equal to 1 (because the button’s name is OK).
4D calls the On Web Authentication Database Method (if it exists), then the *PROCESSFORM* project method is called, which is as follows:

```
If (OK=1)
    If (vEXACT=0) `If the option has not been selected
        vNAME:=VNAME+"@"
    End if
    QUERY([Jockeys];[Jockeys]Name=vNAME)
    vLIST:=Char(1)   `Return the list in HTML
    FIRST RECORD([Jockeys])
    While (Not(End selection([Jockeys])))
        vLIST:=vLIST+[Jockeys]Name+” “+[Jockeys]Tel+”<BR>”
        NEXT RECORD([Jockeys])
    End while
    SEND HTML FILE(“results.htm”)   `Send the list to the results.htm form
        `which contains a reference to the variable vLIST (that is <!--4DVAR vLIST—>)
...
End if
```

URL 4DMETHOD/

Syntax: 4DMETHOD/MyMethod{/Param}
Mode: Contextual. When called from non-contextual mode, switches to contextual mode.
Usage: URL or Form action.

This URL allows you to link an HTML object (text, button...) to a 4D project method in contextual mode. The link will be /4DMETHOD/Method_Name/Param where Method_Name is the name of the 4D project method to be executed when the HTML object is clicked 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). The linked item triggers the execution of the 4D project method through their URLs. The project method can itself display 4D forms, other HTML pages, and so on.

When 4D receives a /4DMETHOD request, the On Web Authentication Database Method (if it exists) is called. If it returns True, the On Web Connection Database Method (if it exists) is called, then the Method_Name method is executed with the /Param string as parameter (in $1).

If you assign /4DMETHOD/Method_Name as form action to a HTML static page, the method is executed when a Submit button of the form is clicked. To submit the HTML form on the 4D side, you need to specify the POST action 4D method that will be executed by 4D after the form is submitted. Refer to the example of the command SEND HTML FILE.
While integrating HTML pages into 4D, you will typically use normal or submit type buttons. The HTML syntax to apply in the form is of the following type:
<FORM ACTION="/4DMETHOD/MethodName" METHOD=POST>
For more information about posted forms, refer to the previous paragraph.

Warning: For a 4D method to be able to be executed using the 4DMETHOD/ URL, it must have the "Available through 4DACTION, 4DMETHOD and 4DSCRIPT" attribute (unchecked by default), defined in the Method properties. For more information on this point, refer to the Connection Security section.

URL 4DCGI/<action>

Syntax: 4DCGI/<action>
Mode: Both.
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 SEND HTTP REDIRECT commands.

When issuing an action, a "response" must be returned, by using commands that send data (SEND HTML FILE, SEND HTML BLOB, etc.).

Warning: Please be sure to execute the shortest possible actions so as not to hold up the browser.

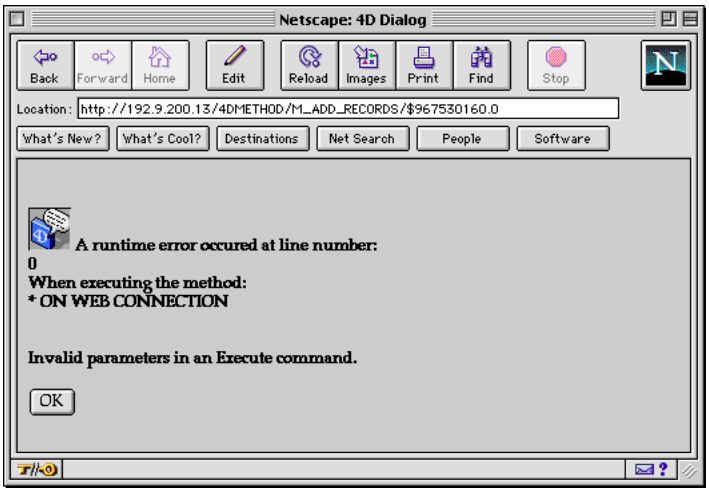
The Text Parameters Passed to 4D Methods Called via URLs

4D sends text parameters to any 4D method called via special URLs (4DMETHOD/, 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 command C_TEXT, otherwise runtime errors will occur while using the Web to access a database that runs in compiled mode.
• 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.

Runtime Errors in Compiled Mode

Let's consider the following example. You execute a method bound to an HTML object using a link and obtain the following screen on your Web browser:



This runtime error is related to the missing declaration of the text $1 parameter in the 4D method that is called when you click on the HTML link referring to that method. As the context of the execution is the current HTML page, the error refers to the "line 0" of the method that has actually sent the page to the Web browser.

Following the example from the section Your First Time with the Web Server, you avoid the problem by explicitly declaring the text $1 parameter within the M_ADD_RECORDS and M_LIST_RECORDS methods:

```
    ` M_ADD_RECORDS project method
C_TEXT($1) ` This parameter MUST be declared explicitly
Repeat
   ADD RECORD([Customers])
Until(OK=0)

    ` M_LIST_RECORDS project method
C_TEXT($1) ` This parameter MUST be declared explicitly
ALL RECORDS([Customers])
MODIFY SELECTION([Customers])
```

After these changes have been made, the compiled runtime errors no longer occur.

Parameters to declare explicitly in the called 4D method

You must declare differents parameters depending on the nature and the origin of the call to a 4D method.

• On Web Authentication Database Method 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)
```

• Method called by the URL 4DMETHOD/
You must declare the $1 parameter:

```
    ` Method called by the URL 4DMETHOD/
C_TEXT($1)
```

• Method called by the URL 4DACTION/
You must declare the $1 parameter:

```
    ` Method called by the URL 4DACTION/
C_TEXT($1)
```

• 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)
```

See Also

Binding 4D objects with HTML objects, GET WEB FORM VARIABLES, Using the Contextual Mode, Your First Time with the Web Server.

4D HTML Tags

The 4D Web server provides you with different HTML tags specific to 4D, which allow you to insert references to 4D variables or expressions, or different types of processing, in static HTML pages sent by the Web server, for example using the SEND HTML FILE and SEND HTML BLOB commands. These pages are called **semi-dynamic pages**.

These tags are inserted as HTML comments (<!--#Tag Contents--> in HTML code). Most HTML editors offer editing facilities to insert comments.

The following 4D HTML tags are available:
- 4DVAR, to insert 4D variables and expressions,
- 4DHTMLVAR, similar to 4DVAR but inserting HTML code,
- 4DSCRIPT, to execute a 4D method,
- 4DINCLUDE, to include a page within another one,
- 4DIF, 4DELSE and 4DENDIF, to insert conditions in the HTML code,
- 4DLOOP and 4DENDLOOP, to make loops in the HTML code.

Compatibility Note: In version 6.0.x of 4D, the notation to use for inserting 4D variables in static pages was square brackets [VarName]. In a converted database, to be able to use the standard HTML syntax (<!--#4DVAR VarName-->), make sure that the option “Use 4DVAR Comments instead of Brackets” in the Preferences dialog box is checked (see section Web Server Settings).

About 4D HTML Tags

Parsing of the contents of semi-dynamic pages sent by 4D takes place when SEND HTML FILE (.htm, .html, .shtm, .shtml) or SEND HTML BLOB (text/html type BLOB) commands are called, as well as when sending pages called using URLs.
However, in non-contextual mode, 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 CACHE STATISTICS command.

Below are the cases where 4D parses the tags contained in the HTML pages sent to the Web browsers:

Sending Conditions	Content analysis of the sent pages	
	Contextual mode	Non-contextual mode
• Page extension (general case):		
.htm, .html, .shtm, .shtml (HTML pages)	X	X
.xml, .xsl (XML pages)	X	X
.wml (WML pages)	X	X
• Pages called via URLs	X	X, except for pages with ".htm" or ".html" extensions
• SEND HTML FILE command call	X	X
• SEND HTML BLOB command call (if the BLOB is "text/html" type)	X	X
• Inclusion by the <!--4DINCLUDE --> tag	X	X
• Inclusion by the {mypage.htm} tag	X	-

In order to be processed by 4D, an HTML comment should have the following format <!--#4D...-->. Please note that some HTML editors automatically add other information within the comment; this can lead to some misinterpretation.
However, other HTML comments such as <!--Beginning of list--> are possible.

If a comment <!--#4D... does not end by -->, the following message "<!--#4D... : --> expected" will be inserted and the analysis will stop at this level (the page will be sent to indicate the error).
It is possible to mix several types of comments. For example, the following HTML syntax is possible:

```
<HTML>
...
<BODY>
<!--#4DSCRIPT/PRE_PROCESS-->                (Method call)
<!--#4DIF (myvar=1)-->                      (If condition)
        <!--#4DINCLUDE banner1.html-->      (Subpage insertion)
<!--#4DENDIF-->                             (End if)
<!--#4DIF (myvar=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: <!--#4DVAR [TABLE]ValNum--></B><BR>
                                                (Field display)
<!--#4DENDIF-->
<!--#4DENDLOOP-->                               (End for)
</BODY>
</HTML>
```

4DVAR

Syntax: <!--#4DVAR VarName--> or <!--#4DVAR 4DExpression-->

The tag <!--#4DVAR VarName--> allows you to insert a reference to the 4D variable or expression VarName anywhere in an HTML page. For example, if you write:
<P>Welcome to <!--#4DVAR vtSiteName-->!</P>
The value of the 4D variable vtSiteName will be inserted in the HTML page.
You can insert a 4D text variable in HTML code, provided its first character is code 1 (i.e., vtHTML:=Char(1)+"...HTML code..."). You can also use the tag 4DHTMLVAR.

You can also insert 4D expressions (not only variables) in 4D HTML comments with the 4DVAR tag. You can directly insert a field content (for example <!--#4DVAR [tableName]fieldName-->) or an item array content (for example <!--#4DVAR arr{1}-->) or a method returning a value (<!--#4DVAR 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 4DVAR depends on the value of the "Available through 4DACTION, 4DMETHOD and 4DSCRIPT" 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, <!--#4DVAR 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.
In case of interpretation error, the inserted text will appear as
"<!--#4DVAR myvar--> : ## error # error code".

Notes:
• You work with process variables.
• You can display a picture field content. In addition (in contextual mode only), you can also display a picture variable content. In both modes, it is not possible to display the content of a picture array item.
• Since HTML is a word processing oriented application, you will usually work with Text variables. However, you can also use BLOB variables (which avoid the 32 000 characters limitation of text type variables). You just need to generate the BLOB in Text without length mode.
• Examples of 4DVAR uses are given in the section Binding 4D objects with HTML objects.

4DHTMLVAR

Syntax: <!--#4DHTMLVAR VarName--> or <!--#4DHTMLVAR 4DExpression-->

This tag allows assessing a variable or a 4D expression and inserting it in a page as an HTML expression. In fact, this tag works exactly the same way as the <!--#4DVAR VarName--> tag when VarName starts with the code 1 (see above).

For example, here are the insertion results of the 4D text variable *myvar* with the available tags:

myvar Value	Tags	Web Page Insertion
myvar:="<B>"	<!--#4DVAR myvar-->	<B>
myvar:=Char(1)+"<B>"	<!--#4DVAR myvar-->	<B>
myvar:="<B>"	<!--#4DHTMLVAR myvar-->	<B>

In case of an interpretation error, the inserted text will be
"<!--#4DHTMLVAR myvar--> : ## error # error code".

Note: Executing a 4D method with 4DHTMLVAR depends on the value of the "Available through 4DACTION, 4DMETHOD and 4DSCRIPT" attribute set in the Method properties. For more information about this, refer to the Connection Security section.

Note: The text variable should be expressed using the ISO Latin-1 character map (for more information, refer to the Mac to ISO command description).

4DSCRIPT/

Syntax: <!--#4DSCRIPT/MethodName/MyParam-->

The 4DSCRIPT tag allows you to execute 4D methods when sending static HTML pages. The presence of the <!--#4DSCRIPT/MyMethod/MyParam--> tag in a static page as an HTML comment forces the execution of the MyMethod method with the MyParam parameter as a string in $1. When loading the home page, 4D calls the On Web Authentication Database Method (if it exists). If it returns True, 4D executes the method.
The method returns text in $0. If the string starts with the code character 1, it is considered as HTML (the same principle is true for the variables).

Note: The execution of a method with 4DSCRIPT depends on the value of the "Available through 4DACTION, 4DMETHOD and 4DSCRIPT" attribute defined in the Method properties. For more information about this, refer to the Connection Security section.

The analysis of the contents of the page is done when either SEND HTML FILE (.htm, .html, .shtm, .shtml) or SEND HTML BLOB (blob of type text/html) is called.
Remember that in non-contextual mode, the analysis is also done when a URL points to a file that has either the ".shtm" or ".shtml" extension (for example http://www.server.com/dir/page.shtm).

Note: In contextual mode, the method is executed in the context.

For example, let's say that you insert the following comment "Today is <!--#4DSCRIPT/MYMETH/MYPARAM-->" into a static 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 "12/31/03"); the expression "Today is <!--#4DSCRIPT/MYMETH/MYPARAM—>" therefore becomes "Today is 12/31/03".
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.

Note: You can insert as many <!--#4DSCRIPT...--> comments as you want in a static page.

Compatibility note: In 4D previous versions, the same tag, 4DACTION, could be used as a URL (for example, http://myserver/4DACTION/meth), or as an HTML comment in a static page (<!--#4DACTION/meth-->). As this could be misleading, starting with 4D version 6.7 the 4DSCRIPT tag replaces the 4DACTION tag. It is used only as an HTML comment (<!--#4DSCRIPT/meth-->) and has the same effect as <!--#4DACTION/meth-->. 4DACTION is now just used for URLs.

4DINCLUDE

Syntax: <!--#4DINCLUDE Path-->

This comment allows the body of another HTML page (indicated by the path parameter) to be included in an HTML page. An HTML page body is included within the <BODY> and </BODY> tags (the tags themselves are not included).
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 mode and file name extension, 4D analyses 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 SEND HTML 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).

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".

Note: In contextual mode, if a page is inserted in a form via a tag {mypage.html} inserted in a static text area, the 4DINCLUDE comments (if any) will be ignored.

Examples

```
<!--#4DINCLUDE subpage.html-->
<!--#4DINCLUDE folder/subpage.html-->
<!--#4DINCLUDE ../folder/subpage.html-->
```

4DIF, 4DELSE and 4DENDIF

Syntax: <!--#4DIF expression--> <!--#4DELSE--> <!--#4DENDIF-->

Used with the <!--#4DELSE--> (optional) and <!--#4DENDIF--> comments, the <!--#4DIF expression--> comment offers the possibility to execute HTML 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.

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-->.

Example

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 <!--#4DVAR vname-->.
<!--#4DELSE-->
No name has been found.
<!--#4DENDIF-->
...
</BODY>
```

4DLOOP and 4DENDLOOP

Syntax: <!--#4DLOOP condition--> <!--#4DENDLOOP-->

This comment allows repetition of a portion of HTML 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 three kinds of conditions:

• **<!--#4DLOOP [table]-->**
This syntax makes a loop for each record from the table current selection in the current process. The HTML 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 HTML code:

```
<!--#4DLOOP [People]-->
<!--#4DVAR [People]Name--> <!--#4DVAR [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 HTML 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 HTML code example:

```
<!--#4DLOOP arr_names-->
<!--#4DVAR 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, 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.

Example
The following HTML code example:

```
<!--#4DLOOP my_method-->
<!--#4DVAR 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
```

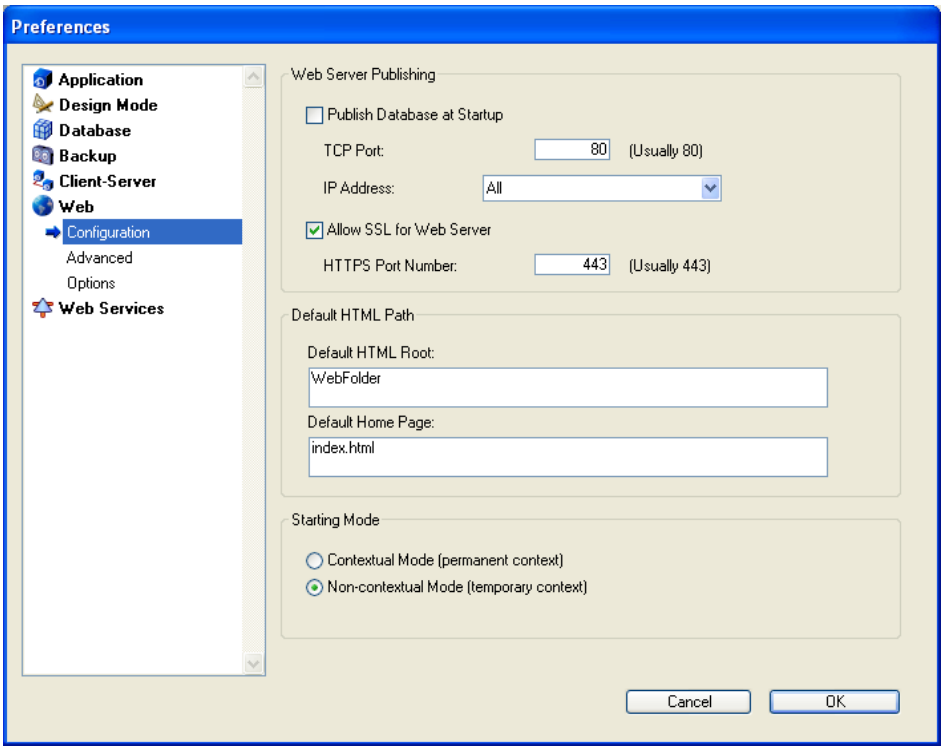
In case of an interpretation error, the text "<!--#4DLOOP expression-->: description" is inserted instead of the contents located between <!--#4DLOOP --> and <!--#4DENDLOOP-->.
The following messages can be displayed:
- Unexpected expression type (standard error);
- Incorrect table name (error on the 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 -->).

See Also

Binding 4D objects with HTML objects, URLs and Form Actions, Using the Contextual Mode.

You can configure the operation of the 4D Web server using the parameters defined in the **Web** theme of the database Preferences. This section describes the parameters of the **Configuration**, **Advanced** and **Options** pages of this theme as well as the “Web Compatibility” section of the **Application/Compatibility** page.

Publishing Page



Publish Database 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.

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. You must use Open Transport version 1.3 or later. This new feature is only available in this version of Open Transport.

2. Open the **TCP/IP** Control Panel.

3. Select the **Manually** option from the **Configuration** pop up menu.

4. 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 Open Transport documentation for more information.

• Installing secondary IP addresses on Windows

To configure a MultiHoming system on Windows:

1. Select the following sequences of commands:
• Windows NT: **Start** menu > **Settings** > **Control Panel** > **Network** Control panel > **Protocols** tab > **TCP/IP Protocol** > **Properties** button > **Advanced**.
• Windows 2000: **Start** menu > **Settings** > **Network and Dial-up Connections** > **Local Area Connection** > **Properties** button > **Internet Protocol (TCP/IP)** > **Properties** button > **Advanced**.
The "Advanced TCP/IP Settings" dialog box appears.
• Windows XP: **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 to do so. For more information, please refer to Windows documentation.

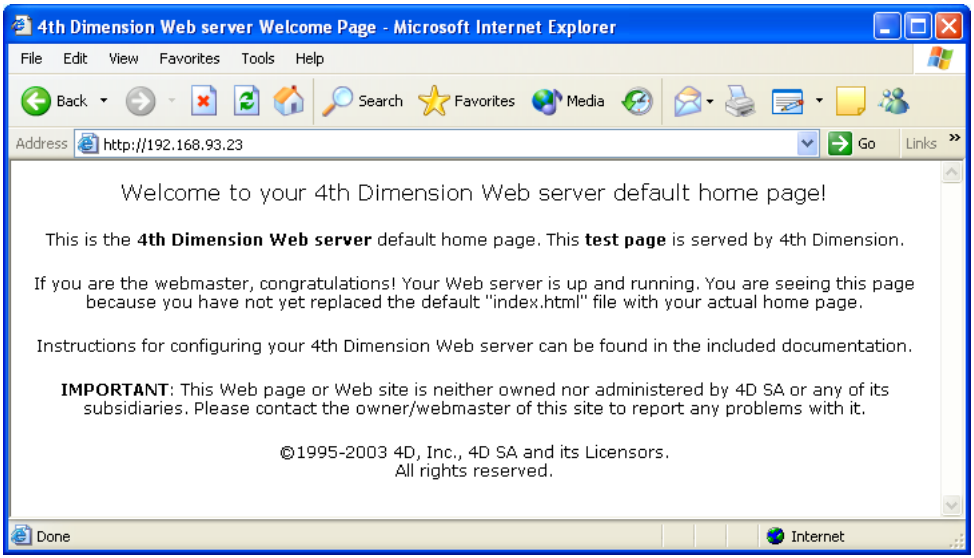
Allow SSL for Web Server
Indicates whether or not the Web server will accept secure connections. This option is described in the Using SSL Protocol section.

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.

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, no matter which mode (contextual or non-contextual) has been defined for the Web sessions. 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 (“/”)

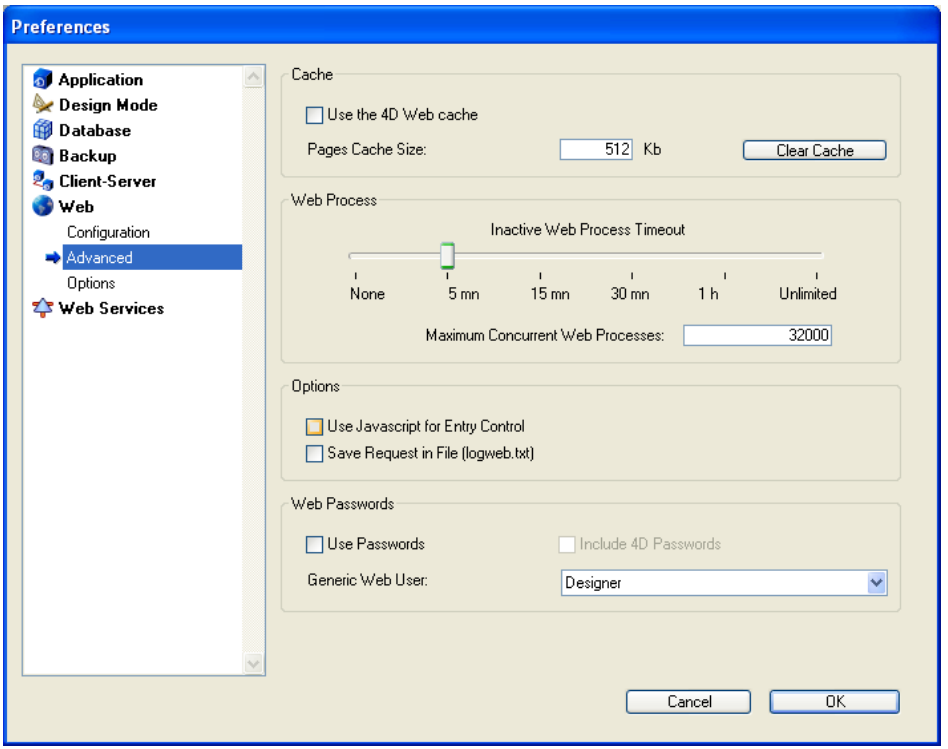
For example, if you want the default home page to be “MyHome.htm”, and it is located in the “Web” folder (itself located in the default HTML root folder of the database), enter “Web/MyHome.htm”.

Note: You can also define a default home page for each Web process by using the routine SET HOME PAGE.
If you do not specify a default custom home page, the operation of the Web server will differ depending on the startup mode:
• If the Web server starts up in non-contextual mode (standard mode), the On Web Connection Database Method is called. It's up to you to process the request procedurally.
• If the Web Server starts up in contextual mode, the current menu bar — by default, menu bar number 1 — is sent as the home page, as in previous versions of 4D.

Starting Mode
Allows you to define the mode in which the Web server will be started. This option is described in the Using the Contextual Mode section.

Reuse Temporary Contexts (only visible with 4D Client)
Allows you to optimize the operation of the 4D Client Web server by reusing Web processes created for processing previous Web requests. In fact, the Web server of 4D Client needs a specific Web process for the handling of each Web request; when necessary, this process connects to the 4D Server machine in order to access the data and database engine. It then 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, 4D maintains the specific Web processes created on 4D Client 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.

Advanced page



Cache for Static Pages

The 4D Web Server has a cache that allows you to load static pages, GIF images, JPEG images (<128 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 CACHE 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 section 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.

Web Process Timeout

Allows you to define the timeout for the Web connection processes (contextual mode only). This option is described in the Using the Contextual Mode section.

Defining the Maximum Number of Web Processes

This option indicates the strictly high limit of **Maximum Concurrent Web Processes** of any type (contextual, non-contextual 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 or an excessive demand of contexts creation.

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.

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.

About the Pool of Web Processes

The "pool" of Web processes allows increasing the reactivity of the Web server in the non-contextual mode. 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. The maximum number of Web processes can also be defined using the SET DATABASE PARAMETER command.

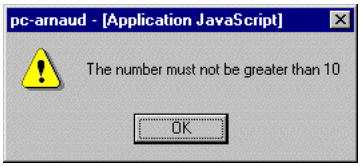
Using Javascript for Data Entry Controls

When this option is checked, in contextual mode a part of the data entry controls can be done on the browsers by using automatic Javascripts.

On the browser, the data entry controls and the data types (fields or variables) to which they can be applied are as follows:

- minimum value (for numeric values)
- maximum value (for numeric values)
- mandatory value (for numeric and alphanumeric values)

Generated Javascripts, which are small in size, display alert dialog boxes without preventing the user from accepting a data entry (it is still 4D's responsibility).
Actually, if a data entry area contains an incorrect value, an alert message is displayed on the browser when the user clicks on a button (OK, Cancel, etc.):



Once the alert dialog box is validated, if the user clicks a second time on the button, the button's action is then taken into account.
Complete data entry control is carried out by the 4D Web server.

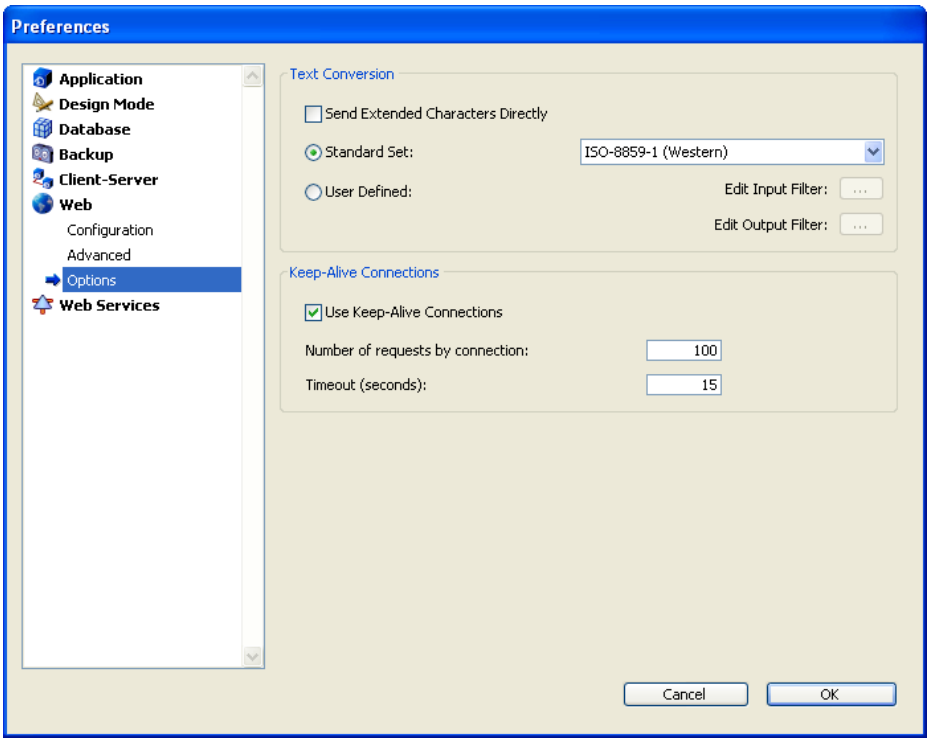
Save Request in File (logweb.txt)

This option enables you to generate a log of requests sent to the Web server in the form of a CLF text file. This option is described in the Information about the Web Site section.

"Passwords" area

Configuration of Web site access protection using passwords. This option is described in the Connection Security section.

Options 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.

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), in contextual and non-contextual mode. 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).

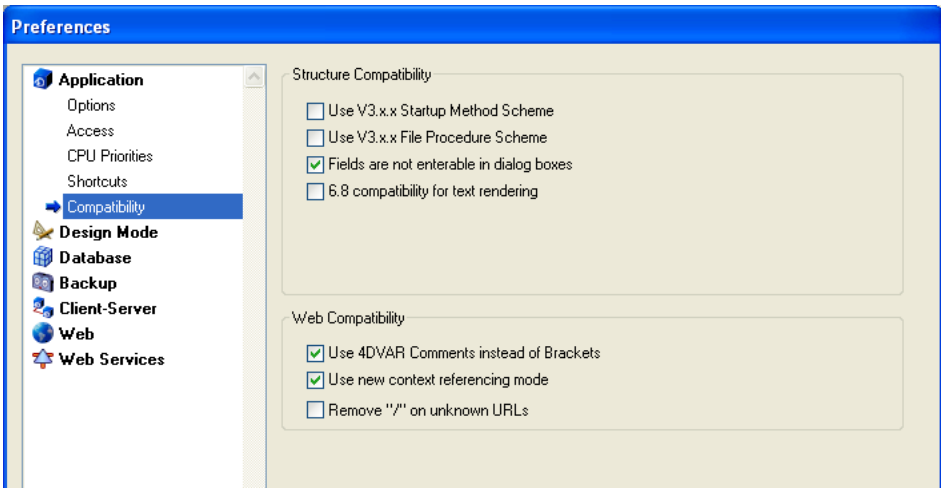
The default value (100) can be increased or decreased depending on the resources of the machine hosting the 4D Web server.

- **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.

Web Compatibility

The **Compatibility** page of the **Application** theme of the Preferences includes options allowing you to adjust the operation of the Web server:



Use 4DVAR comments instead of Brackets

This option allows you to define the notation to use when inserting 4D variables on static pages.
• When the option is checked (default value), the syntax you need to use is the standard HTML notation <!--#4DVAR MYVAR--> (A space character must be inserted between 4DVAR and the variable name).
• When the option is not checked, the syntax you need to use is the notation with square brackets ([MYVAR]) — which is a proprietary solution used in former versions of the 4D Web server.

New Mode to Reference Contexts

When this option is checked (default value), in contextual mode the Web server places the current context number in the basic URL of the documents sent to the browsers.
With the previous system (option not checked), the 4D Web server sends the context number for each element of a page to the browser, which slows down processing.
This option can be unchecked for reasons of compatibility. Keep in mind that after modifying it, you must restart the database in order for the new operation to take effect.

Remove "/" on unknown URLs
In previous versions of 4D, unknown URLs (that corresponded neither to an existing page nor to a 4D special URL) were returned in the On Web Authentication and On Web Connection database methods ($1) without being preceded by the "/" character. This particularity has been removed starting with version 2004.
However, if you have set up mechanisms based on this particularity and wish to keep the previous behavior, you can check the **Remove "/" on unknown URLs** option. By default, this option is checked for converted databases and unchecked for new databases.

See Also

Connection Security, SET DATABASE PARAMETER, SET HOME PAGE, Using the Contextual Mode.

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.

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 the following information in pure text form:
• the number of "hits" (low-level connections),
• the number of contexts created,
• the number of contexts that could not be created,
• the number of password errors,
• the number of pages stored in the cache,
• the percentage of cache used,
• the list of pages and JPEG or GIF files stored in the cache of the static pages (*).

(*) For more information about the cache of static pages and pictures, please refer to section Web Server Settings.

This information can allow you to check the functioning of your server and eventually adapt the corresponding parameters.

Note: The command WEB CACHE STATISTICS allows you to also obtain information about how the cache is being used for static pages.

/4DHTMLSTATS

The /4DHTMLSTATS URL returns, also in pure text form, the same information as the /4DSTATS URL. The difference is that in the last field (contained in the cache), only the list of HTML pages — without the .GIF and .JPEG files — present in the cache is returned.

/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: "Date: Wed, 26 Jan 2000 13:12:50 GMT"
• **Server:** 4D WebStar_D/internal version number
For example: "4D WebStar_D/7.0"
• **User-Agent:** name and version @ IP client address
For example: "Mozilla/4.08 (Macintosh; I; PPC, Nav) @ 192.193.00.00"

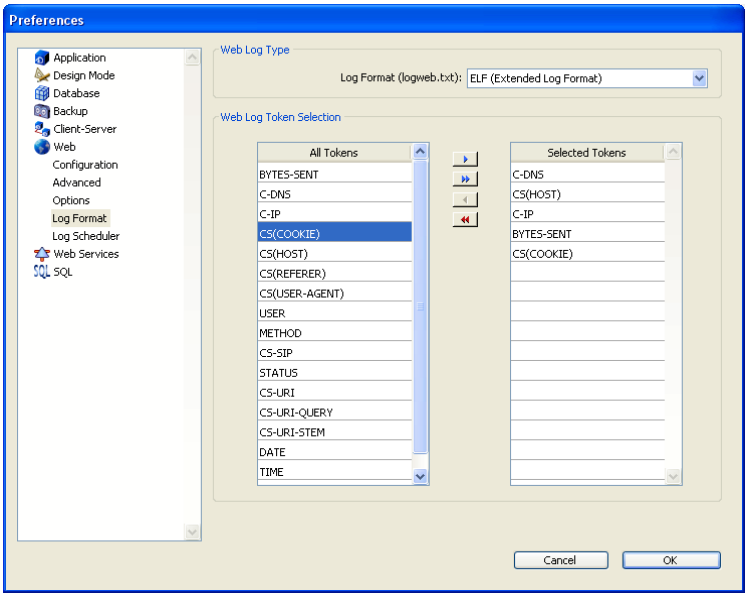
Connection Log File

4D allows you to obtain a log of requests.

This file is named "logweb.txt" and is automatically located:
• with 4D Developer and 4D Server, next to the database structure file.
• with 4D Client, next to the .exe file of the application (Windows) or the software package (Mac OS).

Activation and Format

The activation and configuration of the log file contents is carried out in the Preference of the application on the Web/Log Format page:



Note: The activation and deactivation of the log file of requests can also be carried out by programming using the SET DATABASE PARAMETER command.

The log format menu provides the following options:
• **No Log File**: Whenthis 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). 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

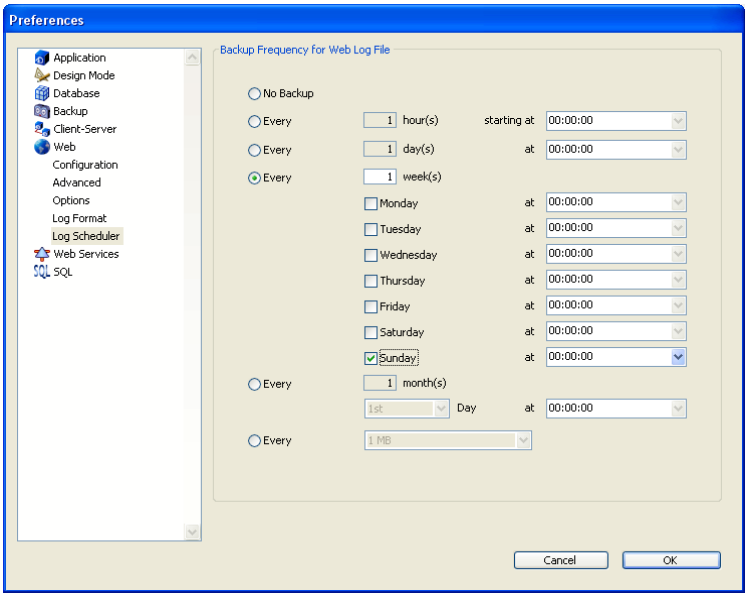
Since a Web log file can become considerably large, it is possible to set up an automatic archiving mechanism. The triggering of a backup can be based on a certain period of time (expressed in hours, days, week or months), or based on the file size; when the set deadline (or file size) is reached, 4D 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 Scheduler page of the application Preferences:



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 in the dataase Preferences.

Runtime Explorer Information

The **Watch** page (“Information” heading) in the Runtime Explorer displays three pieces of information relative to the Web Server:

• **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.

See Also

WEB CACHE STATISTICS, Web Server Settings.

Using the Contextual Mode

The 4D Web server can operate in two different modes: **non-contextual mode** (standard mode) and **contextual mode**. This section describes these two modes and details the particularities of the contextual mode.

Warning: The contextual mode can only be used with 4D Developer and 4D Server. The 4D Client Web server does not support this mode.

Note: The section Your First Time with the Web Server gives a complete example of the publication of a database in contextual mode.

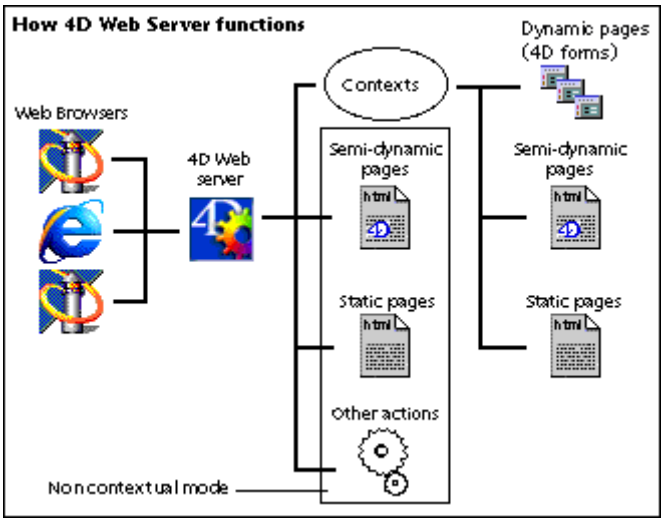
Non-contextual and contextual mode

The 4D Web server uses the **non-contextual mode** by default (disconnected mode). In this mode, the operation of the 4D Web server is comparable to that of standard Web servers: when an HTTP request is received from a browser (URL, posted form, etc.), the server processes the request, then returns a response when necessary (sending a Web page, for example). No specific connection is maintained subsequently between the server and the browser.
In non-contextual mode, the Web server can send static or semi-dynamic pages. Semi-dynamic pages allow you to access data in the database or to carry out any type of processing using special 4D tags that are evaluated at the time the page is sent. Semi-dynamic pages allow you to build, manage and send Web pages whose content originates in whole or in part from processing carried out by 4D. Non-contextual mode generally meets most Web site development needs.

In **contextual mode**, connection to a Web browser causes the creation of a context in which the current selection, its variables, etc. will be placed. In a way, each browser is considered as a 4D Client connecting to the database in Application mode. The context is handled by a specific Web connection process.
This mode allows instant publication of a 4D database on the Web, without it being necessary to create Web pages: 4D manages and sends to the browser dynamic pages stemming from automatic conversion of database menu bars and forms into HTML. In contextual mode, it is still possible to send semi-dynamic or static pages. It is also possible to insert HTML or Javascript code into 4D forms in order to add functions to the pages displayed on the Web.

In addition, in this mode 4D automatically handles simultaneous access to data: when a browser or a 4D Client machine loads a record, 4D locks it for other users in a transparent manner, whether they be browsers or other 4D Client machines. Moreover, 4D allows you to carry out data entry during a transaction with a Web browser, in the same way as with 4D Developer or 4D Client. This system enables the 4D Web server to control the actions of the browsers and to guarantee data integrity.
In return for this ease of publication, the contextual mode includes several constraints:
- Web browsers allowing “surfing” from one Web page to another, one site to another, etc. With a client/server type database, this navigation must be controlled in order to respect the logic of database transactions. Each entry carried out by a user in a record must be validated or cancelled in order not to remain in an uncertain state. The 4D Web server engine contains automatic mechanisms that manage database sessions and contexts. These mechanisms prevent the use of certain standard browser functions (**Reload**, **Previous page**, etc., see next paragraph).
- The connection process in charge of maintaining the context remains active until the timeout of the browser defined in the database Preferences is reached. If, for instance, the browser has left the site in the meantime, the context is then “wasted”.
These constraints mean that the contextual mode is intended rather for Intranet use or for use within the framework of specific Internet applications.

How 4D Web Server functions is summarized in the following figure:

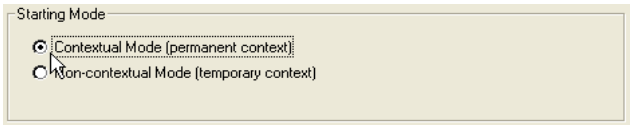


Choice of mode

Choosing the 4D Web server operating mode is carried out as follows:
• On startup of the server, using the **Starting Mode** option of the database Preferences,
• During Web server use, according to the URLs sent or the commands executed.
In fact, some special URLs and certain 4D commands can be used to change the mode. In principle, the current mode remains in use as long as no URL or 4D command causes it to change.

• Defining the contextual mode on startup

By default, the Web server starts in non-contextual mode. You can start the Web server directly in contextual mode. This means that when a user connects to the database, a context is automatically generated.
To specify the non-contextual mode on startup, check the **Contextual Mode (permanent context)** option on the **Configuration** page in the **Web** theme of the database Preferences dialog:



• Commands and URLs that cause the mode to change

During database operation, you can change mode by calling the following elements:
• Change to non-contextual mode:
- SEND HTML BLOB by passing True in the optional noContext parameter
- SEND HTML TEXT by passing True in the optional noContext parameter
- SEND HTTP REDIRECT
- URL beginning with /4DACTION
• Change to contextual mode:
- URL beginning with /4DMETHOD/MyMethod.
When the /4DMETHOD/MyMethod URL is sent, the 4D Web server creates a new context and carries out the following operations:
- The On Web Authentication Database Method is executed (if any),
- The On Web Connection Database Method is executed (if any) — in this particular case, $1 is equal to /4DMETHOD/MyMethod instead of / (slash),
- Finally, the requested method is executed in the newly created context.

Finding out the number of contexts generated

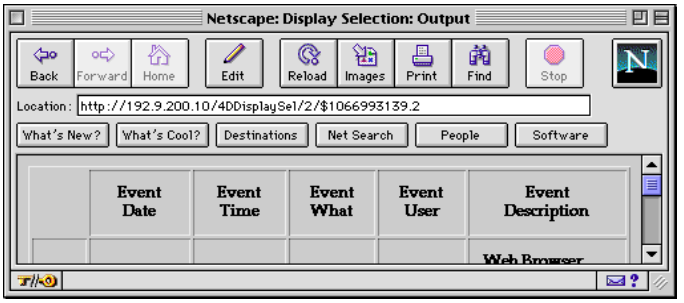
Depending on the actions that they carry out, some Web processes use Web contexts and other do not.

You can find out the number of contexts generated using the PROCESS PROPERTIES command: for any Web process, this command indicates, in the origin parameter, whether it uses a context (-11, Web process with context) or not (-3, Web process without context).

Web Connection Context Management

Web Connection Context ID

The number present in the name of the Web connection process is called the **context ID**, which is randomly generated and uniquely identifies each Web connection. The context ID is maintained on both the 4D and the browser sides during the entire Web connection. In this example, the context ID is 1066993139. In the Web browser window shown here, you can see this number in the URL displayed in the Location area of the browser:



The URLs are automatically maintained by 4D during the whole Web session in contextual mode. Each time an HTTP request is received via TCP/IP, 4D extracts the context ID from the URL, and thereby can redirect the request to the right Web Connection process.

Context IDs:

- Enable 4D to maintain both a Web and Database session over each Web connection.
- Transparently handle multiple concurrent Web connections.
- Prevent future undesirable connections when using bookmarks, because a different context ID is generated at each connection.

Synchronizing Web and Database sessions: Web Connection Subcontext ID

In the window shown above, note that the context ID is followed by a dot and a second number, called the **subcontext ID**. 4D automatically manages and increments this number each time a new 4D-based HTML page is sent to the browser in contextual mode. The subcontext ID is essential to the maintenance of the database session.

Usually, a Web browser includes navigation controls, such as the Back and Forward buttons, History windows, and so on. These controls are useful when you are browsing documents, news, bulletin boards, etc. They are less appealing when you perform a database transaction.

For example, if a Web user is adding a record to a table, you need to know whether or not the data entry is validated, that is, whether or not the Web user clicked the Accept or Cancel buttons of your 4D form. If, at this point, the Web user navigates to other pages, the data entry is left in an uncertain state. To prevent this, 4D uses the subcontext ID to synchronize the Web session on the browser side with the database session on the 4D side.

Each time a form is submitted or an HTTP request is sent to 4D by the browser, if a desynchronization of the Web and database sessions is detected, 4D sends the message “Using browser navigation controls, you left a form requiring data validation. 4D will now return to that form so you can accept or cancel it.” 4D then goes back to the Web data entry page using the subcontext ID.

This synchronization is also essential for the Web Connection process. You need to correctly get out of, for example, an ADD RECORD ([...]) to pursue the execution of your 4D code.

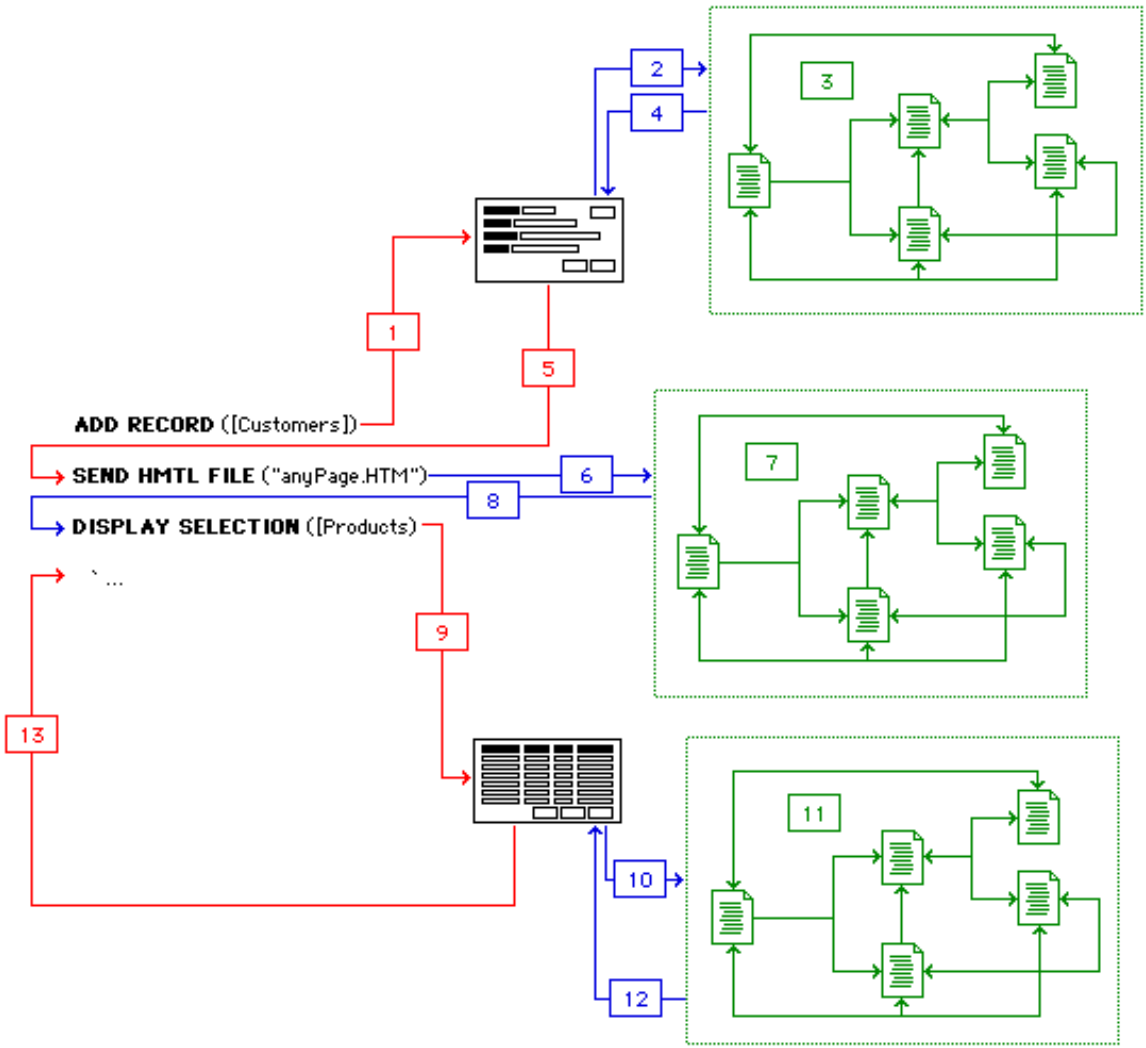
The synchronization is selective. If the current Web page displayed on the browser side is a 4D form (ADD RECORD, DISPLAY SELECTION, DIALOG, etc.), the synchronization will eventuallly occur.
If the current Web page is an HTML page accessed by link from another Web page (sent using the command SEND HTML FILE), then you can navigate freely through the pages.

Given the following piece of 4D code:

```
ADD RECORD ([Customers])
SEND HTML FILE ("anyPage.HTM")
DISPLAY SELECTION ([Products])
```

The following figure details what happens both on 4D and the Web browser during execution.

- Lines in red denote 4D form translations and submissions.
- Lines in blue denote switching back and forth between 4D-based and non 4D-based HTML pages.
- Areas in green denote non 4D-based HTML pages.



Description of the steps

(1) An ADD RECORD is issued. 4D translates the current input form of the table into an HTML page and sends it to the Web browser. If the form is a multi-page form, the standard 4D page navigation buttons allow you to navigate through the pages of the form. This 4D-based navigation is implemented and performed transparently by 4D (via Web form submission).

(2) During data entry (therefore within the ADD RECORD call), a button is clicked and its object method issues a SEND HTML FILE call.

(3) Within the SEND HTML FILE call, if the HTML page includes links, it is possible to navigate through several pages. Eventually, when a SEND HTML FILE("") is issued, the HTML mode is exited.

(4) The object method of the button that was clicked and the data entry initiated by ADD RECORD are executed. Note that steps (2) and (3) can be repeated several times within the data entry.

(5) Finally, the data entry is accepted or canceled, and the Web Connection process is executed.

(6) The next call is a SEND HTML FILE.

(7) This step is analogous to step 3. If the HTML page includes links, it is possible to navigate through several pages. Eventually, when a SEND HTML FILE("") is issued, the HTML mode is exited.

(8) The Web Connection process is executed.

(9) A DISPLAY SELECTION is issued. 4D translates the current output form of the table into an HTML page and sends to the Web browser. During the DISPLAY SELECTION, 4D transparently navigates between the selection page and the display of individual records. 4D also uses MODIFY SELECTION to manage data entry and record locking, via Web form submissions.

(10) During navigation through the selection, a button in the footer area of the form is clicked and its object method issues a SEND HTML FILE call.

(11) This step is analogous to steps 7 and 3. If the HTML page includes links, it is possible to navigate through several pages. Eventually, when a SEND HTML FILE("") is issued, the HTML mode is exited.

(12) The object method of the button that was clicked and the selection display initiated by DISPLAY SELECTION are executed. Note that steps (10) and (11) can be repeated several times during navigation of the selection.

(13) Finally, the selection display is exited and the Web Connection process is executed.

And so on...

Free Web navigation (clicking on the Back or Forward buttons for instance) is possible within any SEND HTML FILE (green areas in the figure above). On the other hand, any 4D-based HTML page (data entry, selection display... including standard dialog boxes such those displayed by CONFIRM or Request) is exited through the use of one of the browser navigation controls, 4D will eventually synchronize the Web sessions and the Database sessions by going back to the Web page whose subcontext ID corresponds to that of the issued 4D command currently being executed on the Web Connection process side.

Web Connection process and Web session

From the user viewpoint, the user's actions on the Web browser side pilot a Web session. From the programmatic viewpoint, the Web Connection process pilots the Web session, not the reverse. The Web browser displays the pages sent by the Web Connection process, which either:

- Executes 4D code, or
- Waits for the submission from the browser of the current Web page.

From a Design viewpoint, the Web Connection process should be seen as a 4D process whose domain of execution is 4D Developer or 4D Server, but whose user interface is remotely echoed on the connected Web browser.

With this in mind, always take into account this duality of the Web Connection process when designing Web database applications in contextual mode. For example:

- During data entry of any kind, the main menu bar is that of the browser, not that of 4D. Within a form, do not rely on the 4D menu bar; it is on the Web server machine, not on the Web browser machine,

- When you design forms to be used on the Web browser, remember that the 4D form set of features is limited to that of HTML (but sometimes with some 4D additions). Do not rely on the whole 4D forms feature set (i.e., object types and form events). For more information on this point, refer to the paragraph “Automatic HTML conversion” below.

- In terms of interprocess communication, CALL PROCESS, when applied to a Web Connection process, has no effects because its current active form is displayed on the Web browser. On the other hand, a Web Connection process can issue a CALL PROCESS toward another 4D process. In addition, interprocess communication can be indifferently performed in both directions using the GET PROCESS VARIABLE and SET PROCESS VARIABLE commands, which do not require a process to have a user interface.

Inactive Web Process Timeout

As explained previously, a Web Connection process in contextual mode is either executing 4D code or waiting for the submission of the Web page currently displayed on the browser side. In the latter case, a Web Connection process will wait for a delay equal to the **Inactive Web Process Timeout,** set on the **Advanced** page (**Web** theme) of the Preferences window or set programmatically using the SET WEB TIMEOUT command.
The scope of the Web Server Connections Timeout setting is the database session. All contextual Web Connection processes are subjected to that value; they are immediately affected if that setting is changed. The default value is 5 minutes.

Note: The command SET WEB TIMEOUT allows you to specify a different Timeout for each Web process.

You can increase or decrease this timeout at your convenience. For example, you can increase the timeout if your application allows Web users to surf to other Web sites via HTML links in the pages served by your database. By increasing the timeout, you enable users to navigate longer within the other Web sites without closing their connections to your databases.

WARNING: There is no way to programmatically stop a Web Connection process. If you specify a long timeout, the process will wait for that delay, even though the Web user may have stopped working with the Web Connection for quite some time. If you specify No Timeout, the Web Connection processes will stop only when the database is exited. However, a Web connection process is automatically aborted as soon as the Web server switches to non-contextual mode.

Tip: Unlike Web Server process, Web Connection processes can be aborted using the Abort command (available in the Runtime Explorer when the Process page is displayed).

Automatic HTML conversion

This paragraph specifies the elements, objects and mechanisms handled automatically during database conversion into HTML by 4D in contextual mode.

Menu Bars

• Each menu bar is translated into one HTML page. Each menu title appears as text only and menu commands associated with 4D methods appear as links to these 4D methods. Menu commands that are only associated with automatic actions appear as text only.
• Clicking a menu item on the Web Browser side starts the execution of the associated 4D method on the Web Connection process side.

Note: When the "Start a New Process" property is assigned to a menu command, the associated method is executed by the 4D Web server in a new Web connection process using the 4DMETHOD URL. In this case, the method of the menu must have the **Available through 4DACTION, 4DMETHOD and 4DSCRIPT** attribute (unchecked by default for new databases). For more information, refer to the Connection Security section.
• The picture associated with a menu bar is placed below the menus on the browser.

Forms

• Objects are translated from top to bottom and from left to right, and they have the same position on the browser as they do in the 4D form. Note, however, that HTML is a word processing oriented application; horizontal objects positions may be different and wrap-around may occur.
• Multi-page forms are supported transparently, including a page zero and inherited forms.
• Automatic actions, when appropriate, are supported transparently.
• Form events (On Load, On Unload, On Clicked and On Timer) are supported. Other events are not supported.
• The Header, Detail, Break and Footer tags are taken into account during calls to DISPLAY SELECTION and MODIFY SELECTION. The Header of the form appears once at the beginning of the HTML page, the detail area is repeated as many times as necessary, and variables (such as buttons) placed in the Footer area appear at the end of the HTML page, just under the automatic selection page navigation links.
• The tips associated with buttons displayed as pictures in the form editor appear on the browser — if the browser allows these tips to be displayed.
• A picture replicated ("Replicated" display) inserted in the (0,0,x,x) coordinates in 4D's form editor is sent as a background picture on the browser. Please note that dark pictures should be avoided.

Note: The 4D Web server uses CSS1 to produce HTML pages with a very similar appearance to the 4D forms themselves. CSS1 (Cascading Style Sheet 1) specifications have been defined by the World Wide Web Consortium (W3C). These style sheets set some characteristics related to the document appearance: font, size, color, title, body, spacing, etc. .CSS documents are sent with the MIME type "text/css". In both contextual or non-contextual mode, these documents are not processed by 4D.
For compatibility reasons, you can modify the Web conversion mode to use for the forms using the SET DATABASE PARAMETER command (selector 8, Web Conversion Mode).

Field Objects

When a 4D form is translated to an HTML page, field objects are translated as follows:

4D Field Type	HTML Object	HTML Markup
Alphanumeric	Text field (*)	<INPUT Type="text" ...>
Text	Text field (*)	<TEXTAREA ...> (**) <INPUT Type="text" ...> (***)
Real	Text field (*)	<INPUT Type="text" ...>
Integer	Text field (*)	<INPUT Type="text" ...>
Long Integer	Text field (*)	<INPUT Type="text" ...>
Date	Text field (*)	<INPUT Type="text" ...>
Time	Text field (*)	<INPUT Type="text" ...>
Boolean	Radio or Check box (*)	<INPUT Type="radio" ...> <INPUT Type="checkbox" ...>
Picture	Image (always non-enterable)	<IMG SRC="..." ...>
Subtable	No HTML support	None
BLOB	No HTML support	None

(*) or text only if non-enterable
(**) If the text value is composed of several lines
(***) If the text value is composed of only one line or is empty

Note: Enterable variables behave like fields of the same type.

Form Objects

When a 4D form is translated to an HTML page, form objects are translated as follows:

4D Form Object	Equivalent HTML Object	HTML Markup
Line	Horizontal Line (1)	<HR>
Rectangle	Rectangle	Managed by CSS1
Oval	No HTML support	None
Rounded Rectangle	No HTML support	None
Static Picture	Image or Image Map (2)	<IMG SRC="..."> <INPUT Type="image" ...>
Group Box	Text	Text with font markups if any
Static Text	Text	Text with font markups if any
Button	Submit button	<INPUT Type="submit" ...>
Default Button	Submit button	<INPUT Type="submit" ...>
Radio Button	Radio button (3)	<INPUT Type="radio" ...>
Check Box	Check Box	<INPUT Type="checkbox" ...>

Pop-up/Drop-down List	Drop-down List	<SELECT ...>...</SELECT>
Combo Box	Drop-down List	<SELECT ...>...</SELECT>
Scrollable Area	Scrollable List (4)	<SELECT ...>...</SELECT>
Tab Control	URL lists (5)	<A HREF="/4DTAB/4DVar.Onglet...">
Invisible Button	See note 2	
Highlight Button	See note 2	
3D Button	See note 2	
Button Grid	See note 2	
Graph	Image (non-enterable)	<IMG SRC="..." ...>
Plug-in	HTML text, Image or Image Map (6)	Text with font markups if any or <IMG SRC="..." ...> or <INPUT Type="image"...>

The following objects are not supported by HTML and therefore are ignored during the translation:
Hierarchical Pop-up menu, Hierarchical List, Subform, Radio Picture, Thermometer, Ruler, Dial, Picture Pop-up Menu, Picture Button, 3D Check Box, 3D Radio Button.

Notes

1. Non-horizontal lines are not supported in HTML; they are therefore ignored.

2. Invisible-like buttons are objects of type Invisible Button, Highlight Button, 3D Button, and Button Grid. If a static picture is not overlapped by an invisible-like button, the picture is translated as a static image. If it is overlapped by at least one invisible-like button, it is translated as a Server-Side Image Map. On the Web browser side, the image is treated as a Server-Side Image Map. On the 4D side, when the submission is received, 4D recalculates the position of the click in order to generate an On Clicked event for the appropriate button, as if the button was actually clicked. Managing invisible-like buttons is therefore quite simple, provided that they overlap with static pictures. You manage these buttons through the Form method or their object methods, as you would do in the regular 4D interface. This also provides you with a very simple way to handle Web Image Mapping. If an invisible-like button does not overlap with any static picture objects, it is ignored during the translation.

3. Radio button grouping is maintained though the translation.

4. Grouped scrollable areas are not supported in HTML. 4D translates them as independent scrollable lists located on the same line.

5. Tab controls (of type array or created by using the values defined in the Object properties) are converted into URL lists.

If the array elements are empty strings, 4D displays 1, 2, 3... on the browser.

6. Plug-in areas are publishable on the Web, by first being converted into HTML, Image or Image Map. This last solution allows you to manage mouse clicks inside the plug-in area (for example, the integrated plug-in 4D Chart is published in an Image Map and the 4D_Pack _AP External clock area is published as an Image). The way in which a plug-in area, which is on a 4D form, is published on the Web depends on the plug-in editor's specifications.

Display Selection / Modify Selection

- The UserSet mechanism is not supported
- An automatic selection paging mechanism is provided by 4D. For more information, see the description of the SET WEB DISPLAY LIMITS command.

4D Commands

While developing a 4D Web database, you may ask what happens when this or that command is called. Will the command take effect on the Web Server machine or on the Web Browser machine? The Web Connection Process is executing on the Web Server machine, but its user interface is remotely echoed on the connected Web Browser. Consequently, for Web database development, the 4D commands can be classified as follows:

Commands that are not affected by execution from within a Web Connection process

A command such as CREATE RECORD works within the executing process; in this case, it creates a record within the Web Connection process. The same applies to commands such as Screen width, which returns the width of the screen on the Web Server machine (the machine on which the process is executing).

Commands that include extra built-in capabilities for transparent Web support

Command Name	Comments
ADD RECORD	Automatic translation of the form, multi-page forms supported
ALERT	Automatic translation of the dialog box
CONFIRM	Automatic translation of the dialog box
DIALOG	Automatic translation of the form, multi-page forms supported
DISPLAY SELECTION	Automatic translation of the form Built-in Web paging mechanism UserSet mechanism is not supported
MODIFY RECORD	Automatic translation of the form, multi-page forms supported
MODIFY SELECTION	Automatic translation of the form Built-in Web paging mechanism UserSet mechanism is not supported
QUERY	Standard Query dialog box supported
QUERY BY EXAMPLE	Automatic translation of the form, multi-page forms supported
Request	Automatic translation of the dialog box
REDRAW	Update of the form displayed on the browser

Commands to use when you know what you want to do

The following commands execute locally on the Web Server machine.

For example, you can invoke the printing of a selection from a Web Browser. However, the printing will be performed on the Web Server machine.

In addition, when a user interface component is involved, it appears on the Web Server machine, i.e., Open document("") vs Open Document("This document"). You should avoid such calls, because the Web Browser will wait for a reply until the dialog box is closed on the Web Server machine. On the other hand, it is perfectly OK to call these routines when no dialog boxes are involved.

Command Name	Comments
Append document	OK, if no file dialog box is invoked
BEEP	Beeps on Web Server machine
Create document	OK, if no file dialog box is invoked
DISPLAY RECORD	Does nothing
EXPORT DIF	OK, if no file dialog box is invoked
EXPORT SYLK	OK, if no file dialog box is invoked
EXPORT TEXT	OK, if no file dialog box is invoked
IMPORT DIF	OK, if no file dialog box is invoked
IMPORT SYLK	OK, if no file dialog box is invoked
IMPORT TEXT	OK, if no file dialog box is invoked
LOAD SET	OK, if no file dialog box is invoked
LOAD VARIABLES	OK, if no file dialog box is invoked
MESSAGE	Messages will appear on Web Server machine
Open document	OK, if no file dialog box is invoked
Open external window	Window opens on Web Server machine
Open resource file	OK, if no file dialog box is invoked
Open window	Window opens on Web Server machine
PLAY	Sound is played on 4D machine
Print form	OK, if no Printing dialog box is invoked
PRINT LABEL	OK, if no Printing dialog box is invoked
PRINT RECORD	OK, if no Printing dialog box is invoked
PRINT SELECTION	OK, if no Printing dialog box is invoked
QUIT 4D	Supported, you can shutdown the Web server remotely
SAVE SET	OK, if no file dialog box is invoked
SAVE VARIABLES	OK, if no file dialog box is invoked
SELECT LOG FILE	OK, if no file dialog box is invoked
SET CHANNEL	OK, if no file dialog box is invoked (documents)
TRACE	Debugger window appears on Web Server machine

Commands Not Supported by Web Connection Processes

Command Name	Comments
ADD DATA SEGMENT	Do NOT call this command from within a Web Connection process This command has not been designed to be used on the Web
ADD SUBRECORD	Do NOT call this command from within a Web Connection process This command has not been designed to be used on the Web
CHANGE CURRENT USER	Do NOT call this command from within a Web Connection process This command has not been designed to be used on the Web
EDIT ACCESS	Do NOT call this command from within a Web Connection process The Passwords window appears on the 4D machine The Browser will wait until the window is closed
GRAPH TABLE	Do NOT call this command from within a Web Connection process This command has not been designed to be used on the Web
MODIFY SUBRECORD	Do NOT call this command from within a Web Connection process This command has not been designed to be used on the Web
ORDER BY	Programmatical support only Standard Order By dialog box not supported on the Web
PRINT SETTINGS	Do NOT call this command from within a Web Connection process The Printing dialog boxes will appear on the 4D machine The Browser will wait until the dialog boxes are closed
QR REPORT	Do NOT call this command from within a Web Connection process The Quick Report window appears on the 4D machine The Browser will wait until the window is closed

Embedding HTML

You can customize the content of 4D forms converted into HTML by embedding HTML code (or Javascript) into the form. The resulting form, on the Web browser side, is a combination of HTML and 4D objects.

Inserting an HTML page using a static text object

A static text object of a 4D form containing, for instance, the string "{page.HTM}", inserts the HTML document "page.HTM" into the 4D form at the place where the text object is located. You insert a document in its entirety (in fact, everything included between the <BODY> and </BODY> tags). You can either use an existing HTML document, or, using language, build a document that you save to disk and to which you refer subsequently.

Note: In some cases, HTML conversion of 4D forms created in version 6.0.x that contain a reference to an HTML document ({mapage.htm}) do not always give the expected result with 4D version 6.7 and later. In this case, it is possible to modify the form conversion mode using the SET DATABASE PARAMETER command.

Inserting HTML code

Any 4D text variable can embed HTML code into a 4D form, if its first character has the code 1 (for example, vtHTML:=Character(1)+"...HTML code...").
You can thus insert pieces of code and, in this case, you can build the HTML code into memory.

File References and URLs

In contextual mode, to insure the maintenance of the database context and subcontext IDs, 4D automatically remaps file references and URLs. For example, 4D remaps all IMG and HREF references to local files.

If you insert your own HTML code into a 4D form using a text variable, you must follow the 4D remapping syntax.

Local GIF files are remapped as "/4DBin/_/GIF_file_pathname", where GIF_file_pathname is the full HTML path name of the GIF file relative to the root of the volume where the file is located.

Example

The following 4D method returns the remapped reference for the pathname received as parameter:

```
    ` WWW Local GIF URL Project Method
    ` WWW Local GIF URL Project ( Text )
    ` WWW Local GIF URL ( Native pathname ) -> URL to local GIF file
C_TEXT($0;$1)
$0:="/4Bin/_/"+HTML Pathname ($1)
```

Note: For details about the method HTML Pathname, see the examples of the command Mac to ISO.

Then, when inserting HTML code into a 4D form using a text variable, you can write:

```
vtHTML:=Char(1)+"<P><IMG SRC="+Char(34)+WWW Local GIF URL("F:\ThisImage.HTM"+
                                        Char(34)+" ALIGN=MIDDLE></P>"+Char(13)
```

This will insert the GIF document in the 4D form at the location of the 4D variable vtHTML.

Important: You only need to write this kind of code to insert custom HTML code into a 4D form. If you just send an HTML page using SEND HTML FILE or if you use a command such as ADD RECORD, remember that 4D transparently translates and remaps the HTML.

The remapping does not change links that have the following protocols:
- http:
- ftp:
- mailto:
- news:
- gopher:
- javascript:
- nntp:
- wais:
- prospero:
- telnet:

See Also

Connection Security, On Web Authentication Database Method, On Web Connection Database Method, SET DATABASE PARAMETER.

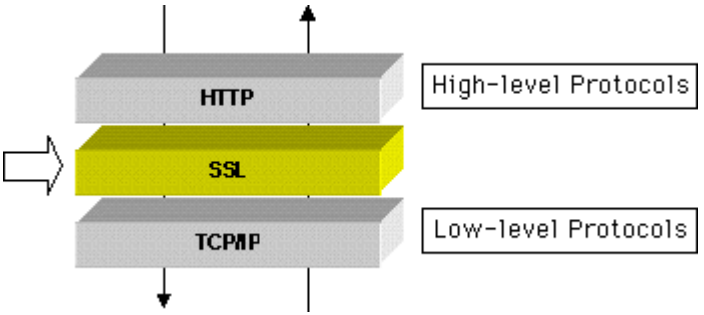
The 4D Web server can communicate in secured mode through the SSL protocol (*Secured Socket Layer*).

SSL Protocol Definition

The SSL protocol 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 SSL protocol is inserted between the TCP/IP layer (low level) and the HTTP high level protocol. SSL has been designed mainly to work with HTTP.

Network configuration using SSL:



Note: The SSL protocol can also be used to secure standard 4D Server client/server connections. For more information, refer to the section Encrypting Client/Server Connections in the *4D Server Reference* manual.

The SSL 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.

SSL uses a public key encryption technique based on a pair of asymmetric keys for encryption and decryption: a public key and a private key.

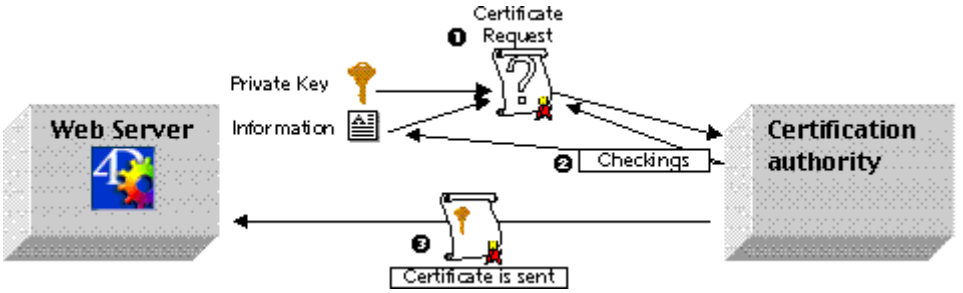
The private key is used to encrypt data. The sender (the web site) does not give it to anyone. The public key is used to decrypt the information and is sent to the receivers (Web browsers) through a certificate. When using SSL with the Internet, the certificate is delivered through a certification authority, 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 4D Web 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 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 SSL 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 Web server machine. 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. This format allows 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). The 4D Web Server accepts all platform-related text formats for certificates (Mac OS, PC, Linux...). However, the certificate must be in PKCS format.

5. Place the “cert.pem” file in the Database structure folder.
The Web server can now work in a secured mode. A certificate is valid between 6 months to a year.

SSL installation and activation within 4D

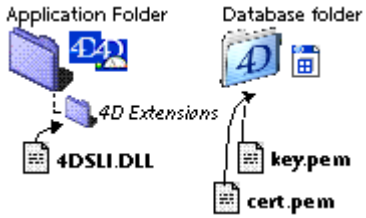
If you want to use the SSL protocol with the 4D Web server, the following components should be installed on the server, at different locations:

• 4DSLI.DLL: Secured Layer Interface dedicated to the SSL management.
This file should be placed in the [4D Extensions] folder of the 4D application that publishes the database.

• key.pem: document containing the private encryption key.
- with 4D Developer or 4D Server, this file must be located in the database folder.
- with 4D Client, this file must be located in the 4D Client application folder.

• cert.pem: document containing the "certificate".
- with 4D Developer or 4D Server, this file must be located in the database folder.
- with 4D Client, this file must be located in the 4D Client application folder.

Files required to implement SSL with 4D Web server (4D Developer and 4D Server):

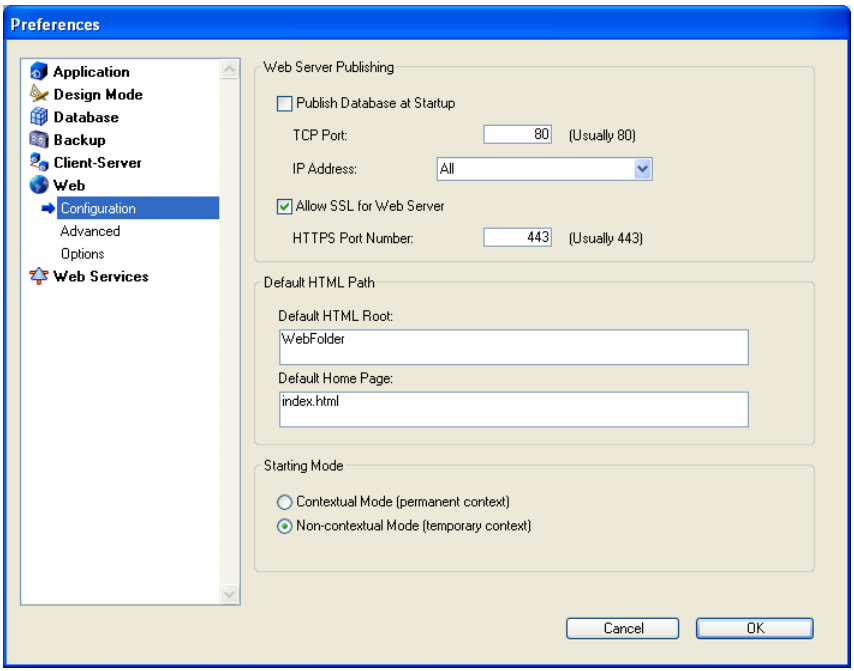


Files required to implement SSL with 4D Web server (4D Client):

Note: 4DSLI.DLL is also necessary to use the encryption commands ENCRYPT BLOB and DECRYPT BLOB.

The installation of these elements makes it possible to use SSL for connections to the 4D Web server. However, in order for SSL connections to be accepted by the 4D Web server, you must "activate" the SSL.

This parameter is accessible on the **Configuration** page of the **Web** theme in the database Preferences:



By default, the SSL connections are allowed. You can uncheck this option if you do not want to use SSL functionalities with your Web server, or if another Web server allowing secure connections is operating on the same machine.

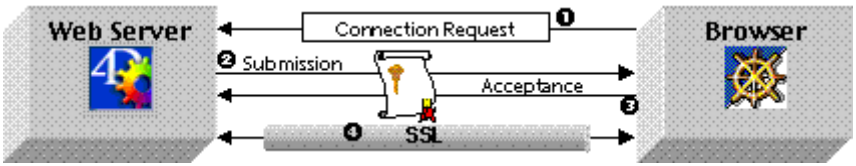
The TCP port dedicated to SSL data exchange is 443 by default. This port number can be modified in the **HTTPS Port Number** area in order, for example, to reinforce the security of the Web server (for more information about this pont, refer to the Web Server Settings section). The TCP port defined in this page of the Preferences is used for standard mode Web server connections.

Note: The other Preferences defined for the 4D Web Server management (password, timeout, cache size, etc.) are applied, regardless of whether or not the server is operating in SSL mode.

Browser connection with SSL

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 depends on current laws in the country of use. The level of encryption offered by 4D Web Server depends on the version of the encryption system library used. By default, 4D provides an "Export" version of the library whereby symetric algorithms are limited to 40 bits.

Management of the connection mode

Using SSL with 4D Web server does not require any specific system configuration. However, you should keep in mind that a SSL Web server can also work in a non-secured mode. The connection mode can switch to another mode if the browser requires so (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. The command Secured Web connection allows you to get the current connection mode.

See Also

DECRYPT BLOB, ENCRYPT BLOB, GENERATE CERTIFICATE REQUEST, GENERATE ENCRYPTION KEYPAIR, Secured Web connection, Web Server Settings.

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 4DVAR 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 4DVAR and 4DSCRIPT 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".

Regardless of the mode applied to the sent documents (contextual or non-contextual mode), 4D analyzes their content and processes their 4DVAR or 4DSCRIPT type tags (if any) in order to generate dynamic XML.

Note: It is not possible to send XML format from a 4D form in contextual mode using a tag such as {mypage.xml} included in a static text.

See Also

Binding 4D objects with HTML objects.

The 4D Web Server supports CGIs (*Common Gateway Interface*). CGIs for Web servers are similar to plug-ins for 4D methods. They are called by the Web server to execute a task and return an answer, i.e. a full Web page or some HTML code inserted in the page sent by the server. CGIs are frequently used to display visitors counters, generate guest books, receive a form-mail, etc. A multitude of CGIs are available today, most of which are freeware.

Note: Originally, CGI was a standard for interfacing external applications with HTTP server. The "CGI" word is now used for the external applications themselves.

The 4D Web Server supports CGIs in two ways:
• 4D Web Server can use CGIs in automatic or manual mode
• 4D Web Server can be interfaced with other HTTP servers using CGIs extensions (Windows only)

Executing CGIs from 4D Web Server

4D supports all types of CGIs, under Mac OS X and under Windows. A CGI can be an application, a PERL script or a DLL interfacing with a Web server.
- **Executables** (.EXE) using the “console” and the environment variables. The source code is usually cross-platform (Windows and Unix). The CGI names are usually written as follows: nnn.exe or nph-nnn.exe. For more information on this kind of CGI, please refer to the http://hoohoo.ncsa.uiuc.edu/cgi/ Internet site.
- **DLL ISAPI**, i.e. extensions for IIS (Internet Information Server). The CGI names are written as follows: nnn.dll or nph-nnn.dll. The DLLs are downloaded once the Web server has been stopped for performance purposes.
For more information on this type of CGI, please refer to the http://www.microsoft.com/iis/ Internet site.
- **PERL scripts** using the “console” and the environment variables. The CGIs require an interpreter to execute. However they are cross-platform (Windows, Unix and Mac OS). Their names are written as follows: nnnn.pl, nph-nnnn.pl, nnnn.cgi or nph-nnnn.cgi.
For more information on this kind of CGI, please refer to the http://www.perl.com/ Internet site.

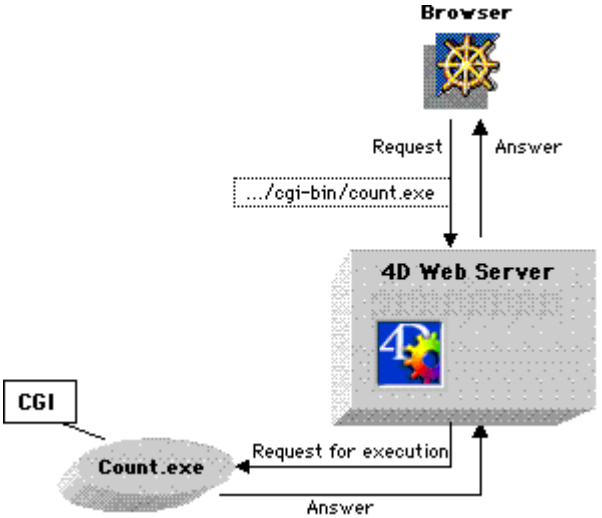
Executing CGIs in automatic mode

An automatic CGI call is made through an URL, an action or a HTML tag inserted within a page according to the task to be done by the CGI. In any case, the HTML string has to contain **/cgi-bin/** followed by the CGI name or possibly a path using the HTML syntax as well as the search string.

For example, an URL “http://195.1.2.3/cgi-bin/search.exe” will trigger the search.exe CGI launch. Similarly, the <IMG SRC="/special/cgi-bin/counter.exe"> marker is placed within a HTML page, the counter.exe CGI will be launched when the page will be sent.

The CGI should be located at the root of the folder named **cgi-bin**. This folder should be placed at the Web server root or in a subfolder. A server can have several **cgi-bin** folders. This folder can contain other files than executable application, but only the latest can be called from a Web client.

Example of installation with a CGI called “Count.exe”:



Below are some examples or locations and matching URLs:

Items location	**Matching URLs**
(Web server root)	
[mybase] folder	
mybase.4db (structure)	(http://195.1.2.3/)
[cgi-bin] folder	
counter.exe	(http://195.1.2.3/cgi-bin/counter.exe)
[Misc] folder	
[cgi-bin] folder	
script.pl	(http://195.1.2.3/Misc/cgi-bin/script.pl)

Executing CGIs in manual mode

Calling CGIs in manual mode requires the use of the SET CGI EXECUTABLE command. In particular, this command lets you execute a CGI without it being visible for the Web user in the URL.
For more information about this point, refer to the description of this command.

Interaction between 4D Web Server and CGIs

A call to a CGI does not modify the 4D environment (selection, variables...).
4D does not limit the response size. However, note that the maximum processing time allocated to a CGI is limited to 30 seconds. After that time, the Web server will return an error.

A CGI is always executed without context, regardless of the calling mode. Note however, that in contextual mode, we strongly recommend that you not use a CGI sending back HTML code, as it might desynchronize the context.

Errors returned by 4D regarding CGI calls

When a call to a CGI generates an error, 4D will return one of the following answers in a standard HTML page:
• *Not found*: 4D does not find the CGI, or a PERL interpreter is missing
• *Forbidden*: the Web client is asking something other than an executable in a [cgi-bin] folder
• *Timeout*: the request has not been processed by the CGI in less than 30 seconds
• *Bad Answer*: 4D was not able to process the CGI answer, or the ISAPI DLL caused an exception
• *Internal Error*: memory full, etc.

Note: When a CGI does not work, check that the execution privileges of the CGI are adequate and that the line feeds in the CGI script are correct.

Information for CGI Developers

This section in mainly intended for programmers who wish to develop specific CGIs for their 4D databases.

• Environment variables

4D defines environment variables in compliance with CGI/1.1 specifications, and the following information:
GATEWAY_INTERFACE: always “CGI/1.1”
SERVER_SOFTWARE: always “4D WebStar_D/version”
SERVER_PROTOCOL: always “HTTP/1.0”
SERVER_PORT_SECURE: contains “1” if the HTTP connection is secure, else “0”.
PATH_TRANSLATED: contains the full path to the HTML server root, and the part of the path following the CGI name. For security reasons, this part cannot contain the character sequences // or ..
Example : Root of the server “C:/web”. For a CGI call such as /cgi-bin/cgi.exe/path, PATH_TRANSLATED value is “C:/web/path”. For a CGI call such /cgi-bin/cgi.exe/../path, 4D returns the error *Forbidden*.
REMOTE_IDENT: user name (if relevant), else undefined.
HTTP_AUTHORIZATION, HTTP_CONTENT_LENGTH and HTTP_CONTENT_TYPE: undefined.
ALL_HTTP and URL are defined in case of ISAPI DLLs calls.
CERT_xxx and HTTPS_xxx are defined if the connection is secured (for DLL only).

Note: The SET ENVIRONMENT VARIABLE command lets you set these variables.

In addition to the standard environment variables, 4D provides text variables FORMVAR_variablename:
- if the request is sent using the "POST" method, these variables are filled with the form entry areas (for example FORMVAR_NAME, FORMVAR_FIRSTNAME...) except for binary fields (INPUT TYPE="FILE"). This system can be used with both “www/url-encoded” and “multipart/form-data” encoded forms.
- if the request is sent using the "GET" method, these variables are filled with the values passed in the request string (for example, in the case of the URL .../cgi.exe?name=smith&code=75, FORMVAR_NAME will get the value “smith” and FORMVAR_CODE will get the value “75”).

This functionality makes form processing easier (it is not necessary to parse strings such as a=1&b=2&...), however the CGI is made 4D-specific.

• Processing of the answers sent by the CGI

If the name of the CGI (Windows executable or PERL script) starts with nph- (*No Parsing Header*), 4D sends the answer “as is” to the Web client. In this case, it is up to the CGI to comply with HTTP rules. Regarding ISAPI DLLs, 4D will never parse the response, whatever the prefix.

Otherwise, 4D will send the HTTP header:
- if “Content-Type” is not specified by the CGI, 4D will always send “Content-Type: text/html”,
- if “Location” is specified, 4D will not take the other elements of the answer into account and will perform a HTTP redirection,
- if “Status” not specified, 4D will send “HTTP/1.0 200 OK”.
4D accepts any kind of line return combination (Windows-CRLF, Mac OS-CR, Unix-LF) in the header of the HTTP answer and will reformat it.

Regarding ISAPI DLLs, 4D accepts asynchrone processings (HttpExtensionProc returns HSE_STATUS_PENDING). A call to ServerSupportFunction (HSE_REQ_DONE_WITH_SESSION) must occur during the next 30 seconds. If the function TerminateExtension is defined, it is always called with the value HSE_TERM_MUST_UNLOAD.

Calling a 4D Web Server using CGIs (Windows only)

4D is provided with two new extensions, 4DISAPI.DLL and NPH-CGI4D.EXE. These extensions have been designed to allow a HTTP server to send requests to a 4D HTTP server. For example, a non secured 4D Web server can be interrogated via another HTTP server, running in secured mode.

Warning: These two extensions are available under Windows only.

• The 4DISAPI extension complies to the specifications defined by ISAPI (*Internet Services Application Programming Interface*). The ISAPI technology has been developed originally by Microsoft® for the IIS server but since it has been made compatible with various HTTP servers such as Netscape®, Apache® or Sambar®.

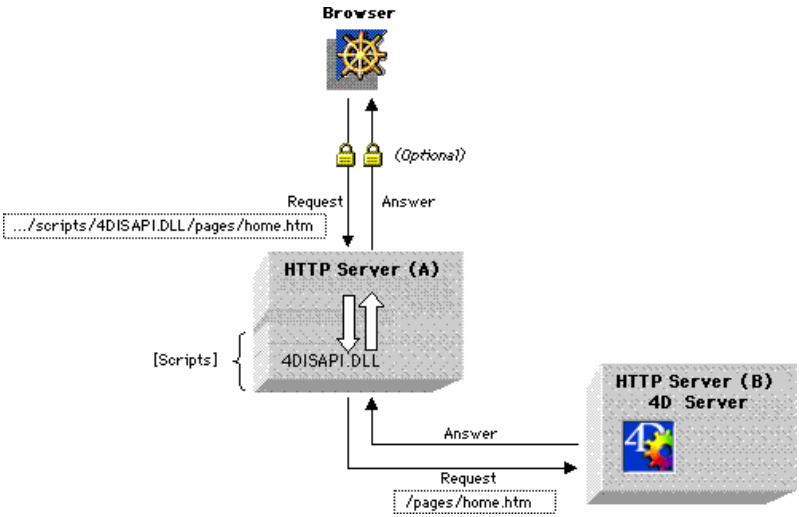
• The NPH-CGI4D.EXE extension complies to the CGI specifications (*Common Gateway Interface*) and can be used with all the CGI compatible servers.

These two extensions work the same way. The CGI compatibility is broadly used with HTTP servers, however the CGI extension performances are usually lower than the ISAPI ones.

How does it work?
These extensions work as follow: HTTP server “A” publishes pages on the Internet, another HTTP server “B” is a 4D Server used in Intranet. To make the two servers communicate, you just have to add the 4DISAPI or NPH-4DCGI extension in the server “A” [Scripts] directory. When a Web browser sends a request to the server A, it transmits it to the server B via the 4DISAPI or NPH-4DCGI extension through the URL (usually, the extension transmits to the server B the URL part located after the call, including the request items). The answer is then sent back to the browser. When the CGI name starts with NPH (*No Parsing Header*), the server does not have to parse the answer HTTP header, the CGI does the formatting. The extensions do not change the HTTP request or answer body.
The initial request sent to the server A can be done in a normal or secured mode (SSL). The communication between the two HTTP servers and the 4DISAPI.DLL extension is done in a non secured mode.

Note: The 4DISAPI and NPH-4DCG extensions are not compatible with 4D Web server contextual mode.

The following figure illustrates this principle:



• The extensions identify the GET, HEAD and POST methods which manage the various status sent back by 4D (*200 OK*, *302 Moved Temporarily*, *404 Not Found*...).

• It is not possible to authenticate at the HTTP level via the 4DISAPI or NPH-CGI4D extension. To do so, an HTML form should be used (which can be done in a secured connection).

Note: The extensions have been designed to receive and send dynamic data and more specifically to post data. The basic Web page service does not offer good performance when the ISAPI or CGI extensions are used.

Installation and configuration
To install 4DISAPI and NPH-CGI4D extensions, copy the 4DISAPI.DLL or NPH-CGI4D.EXE files in the HTTP server [Scripts] folder.

Each installed extension file is provided with a configuration file (.INI file). The .INI file should bear the same name as the extension (for example 4DISAPI.INI). The extension and configuration files should be located in the same folder.

An HTTP server can be defined to target several other HTTP servers. In this case, copy in the HTTP server [Scripts] folder as many extensions as target servers. You just need to rename them (for example, 4DISAPI2.DLL, 4DISAPI3.DLL, etc.). Make sure that a configuration file is associated to each extension and that its name is correct (4DISAPI2.INI, 4DISAPI3.INI, etc.).

A .INI file contains one section: [Forward] which authorizes the following commands:

• TargetServer =
IP name or address of the Web server to target (for example, myserver.net or 192.193.194.195). The line can be left blank for targeting a server by its address if the name resolution is unavailable.
The “localhost” name is identified to the 127.0.0.1 address.
The address 127.0.0.1 is used by default.

• TargetPort =
Port used by the target server (by example, 81). The port number 8080 is used by default.

• Timeout =
Maximum timeout for the server answer (in seconds). The default value is set to 30 seconds.

• Allowed =
Allowed URL list, separated by a comma. For example: /pages, /img to give access only to the URL starting with /pages and /img. To give access to the full site, enter a slash character / (default setting).

• Forbidden =
Forbidden URL list, separated by a comma. For example: /4DMETHOD, /pages2 to forbid the access to the URL starting with /4DMETHOD and /pages2. To give unlimited access, do not enter anything in the list (default setting).

According to the rules stated above, the access to the following URLs will be given or not:

/pages/document.html	accessible
/pages1/onepage.html	accessible
/www/picture.gif	not accessible
/pages2/mypage.html	not accessible
/4dmethod/myproc	not accessible

If a forbidden URL is called, the extension directly returns the error "*HTTP/1.0 403 Forbidden*".

Using the extensions

4DISAPI and NPH-CGI4D extensions support the following URLs:

• **4D call** (4D will receive only the URL part located after the extension name):
http://server-address/cgi-bin/4disapi.dll/[Path]
http://server-address/cgi-bin/nph-cgi4d.exe/[Path]

• **Checking of the extension** (echo of the request):
http://server-address/cgi-bin/4disapi.dll/~~echo
http://server-address/cgi-bin/nph-cgi4d.exe/~~echo

• **Information about the extension** (technical support):
http://server-address/cgi-bin/4disapi.dll/~~info
http://server-address/cgi-bin/nph-cgi4d.exe/~~info
The following information is returned:
name and version of the extension, for example "Script name: 4disapi.dll (6.7.0b1.2)"
name and version of the server calling the extension, for example "Server software: 4D_WebStar_D/6.7"
version of the HTTP protocol, for example "Server protocol: HTTP/1.0"
version of the CGI protocol, for example "Gateway interface: CGI/1.1"

• **Checking of the target server** (is the server available?) :
http://server-address/cgi-bin/4disapi.dll/~~target
http://server-address/cgi-bin/nph-cgi4d.exe/~~target
The answer is either:
"Good: target server reached.": the target server answered (whatever the contents of the answer).
or "Bad: target server not reached.": the target server cannot be reached or did not answer.

In this case, the fail can be caused by:
- the configuration file is missing;
- the target address or port is incorrect;
- the target server is out;
- the target server did receive the request but is unable to answer.

Note concerning 4D WebSTAR: 4D WebSTAR® is one of the most popular Web servers on Mac OS. Various interaction possibilities between 4D and 4D WebSTAR have been developed, in particular the 4D WebSTAR plug-in called 4D Link. For more information about this plug-in, please refer to the 4D WebSTAR documentation.

See Also

SET CGI EXECUTABLE, SET ENVIRONMENT VARIABLE.

GET HTTP BODY (body)

Parameter	Type	Description
body	BLOB | Text ←	Body of the HTTP request

Description

The 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 only operates in non-contextual mode. It can be called using a Web database method (On Web Authentication Database Method, On Web Connection Database Method) or any Web method executed in non-contextual mode.

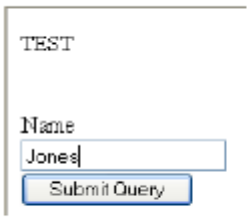
In body, you can pass a variable or a field of the BLOB or Text type. Keep in mind that it is generally preferable to use the BLOB type since the number of characters is not limited. The Text type, on the other hand, is limited to 32,000 characters; if you exceed this amount, any excess data received will be truncated.

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:

Form



Body

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
<title>Test page</title>
</head>

<body>
TEST
<FORM ACTION="/4DACTION/Test4D2004" METHOD=POST>
<br>Name</br>
<input type="text" value="Enter your name" name="vName">
<br>
<input type="submit">
</FORM>
</body>

</html>
```

Here is the *Test4D2004* method:

```
C_BLOB($request)
C_TEXT($requestText)

GET HTTP BODY($request)
$requestText:=BLOB to text($request;UTF8 Text without length)
SEND HTML FILE("page.html")
```

Note: This method is declared "Available through 4DACTION, 4DMETHOD and 4DSCRIPT" in its properties.

When the form is submitted to the Web server, the *$requestText* variable receives the text of the HTTP request body.

See Also

GET HTTP HEADER.

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 GET HTTP HEADER command returns either a string or two arrays, containing the HTTP header used for the currently processed request.

This command works only in non-contextual mode. It can be called from within any method (On Web Connection Database Method or On Web Authentication Database Method, method called by '/4DACTION'...) executed in a Web process in a non-contextual mode. If GET HTTP HEADER is called in contextual mode, it returns empty strings.

• **First syntax: 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: 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)
$0:=""
GET HTTP HEADER($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)
        SEND HTML FILE("index_fr.html")
    : ($language="@sp@")    `Spanish (see list ISO 639)
        SEND HTML FILE("index_es.html")
    Else
        SEND HTML 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")
        SEND HTML FILE("home_site1.com")
    : ($host="www.site2.com")
        SEND HTML FILE("home_site2.com")
    Else
        SEND HTML FILE("home_site.com")
End case
```

See Also

GET HTTP BODY, SET HTTP HEADER.

GET WEB FORM VARIABLES (nameArray; valueArray)

Parameter	Type		Description
nameArray	Text Array	←	Web form variable names
valueArray	Text Array	←	Web form variable values

Description

The GET WEB FORM 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, checkbox, radio button, pop up menu, choice list...

Note: Regarding checkboxes, the variable name and value are returned only if the checkbox has been actually checked.

This command is valid for non-contextual mode or in contextual mode, 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)
GET WEB FORM 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.

See Also

Binding 4D objects with HTML objects, URLs and Form Actions.

OPEN WEB URL (url{; *})

Parameter	Type		Description
url	String	→	Startup URL
*	*	→	If specified = URL is not translated, If omitted = URL is translated

Description

The OPEN WEB URL command launches your default Web browser and opens it on the URL passed in the url parameter.

If there is no browser on the volumes connected to the computer, this command has no effect.

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 type "http://www.server.net/page.htm?q=something".

Note: This command does not work when called from a Web process.

Examples

1. When the following line of code is executed:

```
OPEN WEB URL("file:///D:/web file.htm")
```

The Web browser is launched and the URL is translated into "file:///D%3A/web%20file.htm".

2. When the following line of code is executed:

```
OPEN WEB URL("file:///D:/web file.htm";*)
```

The Web browser is launched and the URL remains "file:///D:/web file.htm"

3. When opening files locally, it is possible to use system separators in the url parameter:

```
$ref:=Open document("";"";Get Pathname)
CLOSE DOCUMENT($ref)
OPEN WEB URL("file:///"+document)
```

4. The following line of code launches the browser and connects it to 4D's home page:

```
OPEN WEB URL("http://www.4d.com/")
```

PROCESS HTML TAGS (inputData; outputData)

Parameter	Type	Description
inputData	BLOB | Text →	Data containing HTML tags to process
outputData	BLOB | Text ←	Processed data

Description

The PROCESS HTML TAGS command causes the processing by 4D of 4D HTML tags contained in the inputData parameter (field or variable of the BLOB or Text type) and returns the resulting data in outputData.

This command lets you carry out processing on tagged HTML code without it being necessary for the Web server to send an HTML page using a command of the SEND HTML BLOB type or that a page suffixed ".shtml" be requested via a URL. It is not even necessary for the 4D Web server to be started.

Pass the data containing the tags to be processed in the inputData parameter. This parameter can be a field or variable of the BLOB or Text type. Keep in mind that it is generally prefereable to use the BLOB type since the number of characters is not limited (32,000 limit for the Text type).

All the HTML tags of 4D are supported (4DVAR, 4DSCRIPT, 4DLOOP, etc.), regardless of the Web server operating mode (contextual or non-contextual) — and even when it is not started.

Note: When using the 4DINCLUDE tag outside the framework of the Web server (Web process):
• with 4D Developer or 4D Server, the default folder is the folder containing the database structure file,
• with 4D Client, the default folder is the folder containing the 4D Client application.

After command execution, the outputData parameter receives the data of the inputData parameter, along with the result of the processing of any 4D HTML tags that it contains, when applicable. If inputData does not contain any 4D HTML tags, the contents of outputData is identical to that of inputData.
The outputData parameter may be a field or a variable, but it must be of the same type as that of the inputData parameter.

This command makes it possible to store the values resulting from the processing of HTML tags in the database before they are sent.

It also permits the parsing of 4D HTML tags apart from the use of the Web server. In particular, 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.

Example

The following example shows how this command works:

```
C_BLOB($in)
C_BLOB($out)
C_TEXT($in_text)
C_TEXT(Var)
C_TEXT(VarHTML)

Var:="<B>"
$in_text:="<p><!--#4DVAR Var→</p>"
TEXT TO BLOB($in_text;$in;UTF8 Text without length)
PROCESS HTML TAGS($in;$out)
VarHTML:=BLOB to text($out;UTF8 Text without length)
    ` HTMLvar contains <p>&lt;B&gt;</p>
```

See Also

4D HTML Tags.

Secured Web connection

Secured Web connection → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	True = the web connection is secured. False = the web connection is not secured.

Description

The Secured Web connection command returns a Boolean indicating if the 4D Web server connection was done in secured mode through SSL (the request starts with “https:” instead of “http:”).

- If the connection is made through 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 SSL protocol, refer to section Using SSL Protocol.

This command allows, for example, denying connections made in a non-secured mode (if any).

See Also

GENERATE CERTIFICATE REQUEST, Using SSL Protocol.

SEND HTML BLOB (blob; type{; noContext})

Parameter	Type		Description
blob	BLOB	→	BLOB to send to the browser
type	String	→	Data type of the BLOB
noContext	Boolean	→	True = Switch to non contextual mode False = Keep current mode

Description

The SEND HTML 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 SEND HTML BLOB command allows you to therefore send all types of documents, the Intranet clients automatically open the associated application.

Note: If the BLOB is of type “text/html” (.htm, .html, .shtm, .shtml), it is translated and analyzed as an HTML file. In this case, when used in contextual mode, SEND HTML BLOB works exactly as SEND HTML FILE. That is, a 4D method that issues a SEND HTML BLOB("") call should be called in one of the HTML pages, in order to terminate the original SEND HTML BLOB call. For more information, refer to the SEND HTML FILE command description.

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
.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
.z	application/x-zip

.zip	application/x-zip
.gz	application/x-gzip
.tar	application/x-tar

Note: For more information, go to http://www.iana.org and look for “Protocol Numbers and Assignment Services” topics.

The noContext parameter allows you to tell the 4D Web server that you want to switch from contextual mode to non-contextual mode. In this case, pass True to noContext.
If the parameter is omitted or contains False, the current mode is used.
The references to 4D variables and 4DSCRIPT type tags in the page are always parsed, whatever the mode.

Example

Refer to the example of the routine PICTURE TO GIF.

See Also

SEND HTML FILE.

SEND HTML FILE (htmlFile)

Parameter	Type		Description
htmlFile	String	-->	HTML Pathname to HTML file or empty string for terminating SEND HTML FILE

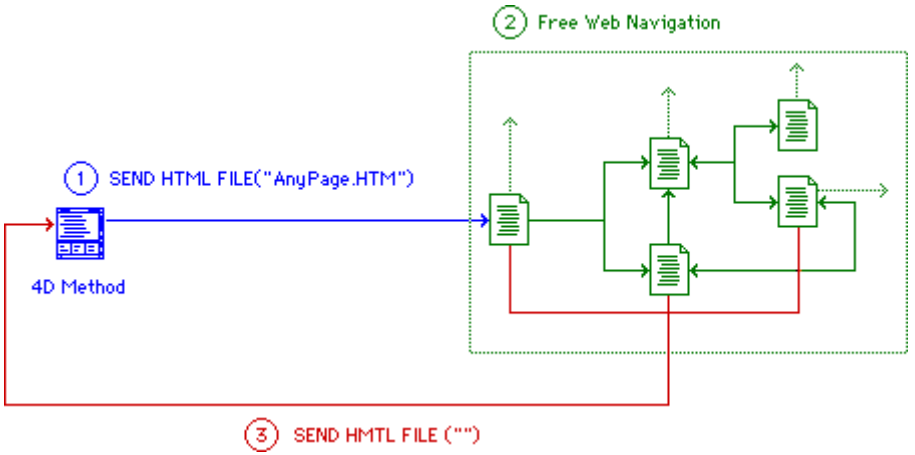
Description

The SEND HTML FILE command sends, to the Web browser, the Web page stored in the HTML document whose pathname you pass in htmlFile.

By default, 4D looks for the HTML document within the HTML root folder, defined in the application Preferences.

This command will only acccept path names in HTML syntax as a parameter: names of directories or folders must be separated with a slash ("/") regardless of the platform.
If you specify an invalid HTML pathname, 4D sends the message “The requested HTML page could not be found” to the Web browser.

The alternate syntax SEND HTML FILE(""), in which you pass an empty string in hmtlFile, allows you, in contextual mode, to terminate the call to SEND HTML FILE, which initiated the HTML mode. This is illustrated in the following diagram:



1. In contextual mode, a 4D Method (Project, Object or Database) issues a call to SEND HTML FILE, sending an HTML document to the browser.
2. The initial Web page sent to the browser may have HTML links to other Web pages or can itself refer to 4D Methods that call SEND HTML FILE to send other Web pages. These other pages may have links or refer to 4D Methods for accessing other pages, and so on. While navigating through the Web pages, you can also use browser's navigation controls, such as the Back button.

3. Any of the Web pages can include references to a 4D method that issues a SEND HTML FILE("") call. This call terminates the SEND HTML FILE call that initiated the whole thing, and you go back, pursuing the execution 4D Method that originally started the free Web navigation.

Once SEND HTML 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 SEND HTML 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 4DSCRIPTS type tags in the page are always parsed, whatever the mode.

Examples

1. 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 :

```
SEND HTML FILE ("HTM/MyPage.HTM")
```

2. Example in contextual mode:during a 4D Web session, you are adding records using a 4D form. In this form, there is a bHelp button, whose object method is as follows:

```
` bHelp button Object Method
SEND HTML FILE ("Help.HTM")
```

Starting from the Help.HTM document, you can freely navigate between numerous HTML pages which implement the database Help system for your Web site. In each page, you have a submit button titled Done, which allows you to go back to data entry.

To do so, each of the HTML documents must contain the definition of this submit button:

```
<!-- bDone submit button -->
<P><INPUT TYPE="submit" NAME="bDone" VALUE="Done"></P>
```

as well as the definition of the form post action:

```
<!-- Execute the 4D htm_Help_Done when a submit button is hit -->
<FORM action="/4DMETHOD/htm_Help_Done" method="POST">
```

On the 4D side, the project method htm_Help_Done terminates the SEND HTML FILE initiated by the bHelp button:

```
   ` htm_Help_Done Project Method
SEND HTML FILE ("")
```

The call to SEND HTML FILE in the object Method of the bHelp button is the last line of the method. When the method is completed, you return to data entry.

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.

See Also

Binding 4D objects with HTML objects, SEND HTML BLOB, Your First Time with the Web Server.

SEND HTML TEXT (htmlText{; noContext})

Parameter	Type		Description
htmlText	Text	→	HTML text field or variable to be sent to the Web browser
noContext	Boolean	→	True = Go to non contextual mode False or omitted = Remains in the current mode

Description

The SEND HTML TEXT command directly sends HTML formatted text data.

The htmlText parameter contains the data to be sent. As 4D does not check the parameter content, make sure that the HTML encoding is correct. The text variable should be expressed using the ISO Latin-1 character map.

Note: This command is similar to the SEND HTML BLOB command using a BLOB with a "html/txt" type.

The noContext parameter indicates to the 4D Web server that you want to switch from the contextual mode to the non-contextual mode when executing the command.
In this case, pass True in the noContext parameter if you want to use the non-contextual mode. Pass False or nothing if you want to use the current mode.

The references to the 4D variables and 4DSCRIPT type tags (if any) in the text are always analyzed, regardless of the mode.

Example

The following method:

```
TEXT TO BLOB("<html><head></head><body>"+String(Current time)+"</body></html>";
                                                    $blob;;UTF8 Text without length)
SEND HTML BLOB($blob;"text/html")
```

... can be replaced by the single line:

```
SEND HTML TEXT("<html><head></head><body>"+String(Current time)+
                                                    "</body></html>")
```

See Also

Mac to ISO, SEND HTML BLOB.

SEND HTTP RAW DATA (data{; *})

Parameter	Type		Description
data	BLOB	→	HTTP data to send
*	*	→	Send chunked

Description

The SEND HTTP RAW DATA command lets the 4D Web server send “raw” HTTP data, which can be chunked. It only operates in non-contextual mode.

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://...
```

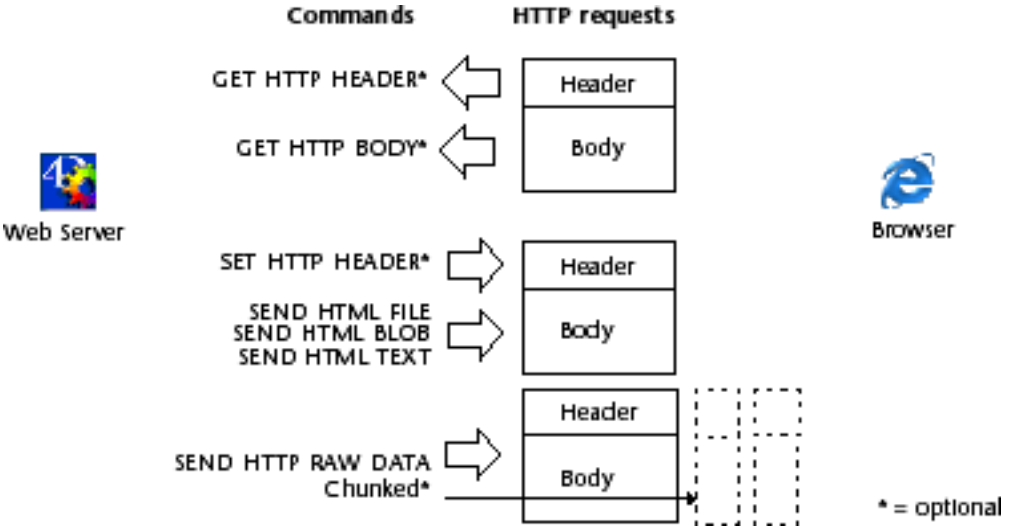
4D will complete the request by adding HTTP/1.1 302.

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 SEND HTTP RAW DATA statements that follow the execution of SEND HTTP 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)
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 new 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 SEND HTTP 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 SEND HTTP RAW DATA 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)
    SEND HTTP RAW DATA($my_blob;*)
End for
```

See Also

GET HTTP BODY, GET HTTP HEADER.

SEND HTTP REDIRECT (url{; *})

Parameter	Type		Description
url	String	→	New URL
*	*	→	If specified = URL is not translated, If omitted = URL is translated

Description

The 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: SEND HTTP REDIRECT ("/MyPage.HTM").

When this command is called in contextual mode, the Web process is aborted just after being executed. The command prevails over commands that send data (SEND HTML FILE, SEND HTML 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.

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 part (or subroutine) that manages the non-contextual mode, 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
         SEND HTTP REDIRECT("/4dcgi/rech?"+name)
      Else
         `Else return to the beginning page
         SEND HTTP REDIRECT("/page1.htm")
      End if
      ...
   : ($1="/4dcgi/rech?@") `If the URL has been redirected
      ... `Put the request code here
End case
```

SET CGI EXECUTABLE (url1{; url2})

Parameter	Type		Description
url1	String	→	Access URL
url2	String	→	Access URL

Description

The SET CGI EXECUTABLE command is used to execute a CGI without it being visible to the Web user in the URL. This command can be used in particular in the On Web Authentication Database Method to determine, for example, which CGI to execute. It operates both under Mac OS X and Windows.

Note: For more information about CGIs, refer to the Using CGIs section.

In url1, pass the access URL for the CGI to be executed. For example, if you write SET CGI EXECUTABLE("/myfile.pl"), the 4D Web server will execute the CGI myfile.pl — this application must be located in the default folder of the Web server.
If you pass an empty string ("") in url1, 4D will execute the CGI specified in the URL sent by the browser directly, where applicable.

In the optional url2 parameter, pass the access URL for the file that you want to be processed by the CGI. For example, if you write SET CGI EXECUTABLE("cgi-bin/Perl2.cgi";"Perl2.pl"), the Web server will execute the CGI Perl2.cgi (located in the cgi-bin folder) by passing it the Perl2.pl file.
If you pass an empty string ("") in url2, 4D will pass the file specified in the URL sent by the browser to the CGI for processing. This mechanism is used more particularly by PHP. Example: SET CGI EXECUTABLE("/cgi-bin/php";"").

If the access URL indicated by the command is incorrect, the browser will display the “File not found” error page.

Keep in mind that the SET CGI EXECUTABLE command does not return an error directly. This command only sets a “current value” that will be used subsequently when the CGI is called. In the event of multiple calls with this command, only the value indicated by the last call will be used.

Example

In this example, the example.php file, which is not located in the cgi-bin folder, is processed by the CGI Perl2.cgi, located in the cgi-bin folder:

```
SET CGI EXECUTABLE("/cgi-bin/Perl2.cgi";"example.php")
```

See Also

Using CGIs.

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 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.
To stop sending homePage as home page for the current Web process, execute SET HOME PAGE with an empty string ("") passed in homePage.

Note: 4D also allows you to define a default home page in the Preferences dialog box. In this case, the page applies to all the Web connections whatever the Web server's startup mode (contextual or non-contextual).

See Also

Web Server Settings.

SET HTML ROOT

Web Server

version 11 (Modified)

SET HTML ROOT (rootFolder)

Parameter	Type		Description
rootFolder	String	→	Pathname of Web server root folder

Description

The SET HTML ROOT 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 Preference 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

Note: The Web server will need to be restarted in order for the new root folder to be taken into account.

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.

See Also

ON ERR CALL.

Error Handling

If you specify an invalid pathname, an OS File manager error is generated. You can intercept the error with an ON ERR CALL method.

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 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 in non-contextual mode.
This command allows you to manage “cookies”.

Two syntaxes are available for this command:

• **First syntax**: 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)
$vTcookie:="SET-COOKIE: USER="+String(Abs(Random))+"; PATH=/"
SET HTTP HEADER($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**: 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"

valueArray{1}:="HTTP/1.0" *
valueArray{2}:="200 OK" *
valueArray{3}:="C=HELLO"
```

* 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.

If several SET HTTP HEADER calls occur in the same Web process, only the last call is taken into account.

The Server, Date and Content-Length fields are always set by 4D.

See Also

GET HTTP HEADER.

SET WEB DISPLAY LIMITS

Web Server

version 6.0

SET WEB DISPLAY LIMITS (numberRecords{; numberPages{; picRef}})

Parameter	Type		Description
numberRecords	Number	→	Maximum number of records to display in each HTML page
numberPages	Number	→	Maximum number of page references at bottom of each HTML page
picRef	Number	→	Picture reference number for full page record button

Description

The SET WEB DISPLAY LIMITS command modifies the way 4D displays a selection of records on the Web browser side when you call DISPLAY SELECTION or MODIFY SELECTION. This command only operates in contextual mode.

When you display a selection of records using 4D, the program does not load all the records of the selection; it only loads (from the disk) the records that are visible in the window at one time. In doing so, although you create a selection of thousands of records, displaying them is quite fast. Then, if you scroll or resize the window, 4D loads the records appropriately.

On the Web, 4D divides the selection of records to be displayed in pages. Without a paging scheme, a selection of thousands of records would result in thousands of records going over the Internet or your Intranet to be displayed in only one Web page. It also would take quite some time to download these records, and your Web browser would more than likely run out of memory.

By default, 4D displays the first 20 records of the selection and includes, at the end of each HTML page, 20 links to the first 20 pages of the selection. This means that, by default, you can browse the first 400 records of the selection by clicking on the page links located at the end of each selection page. Note that this paging system is transparent to your coding; everything happens within the call to DISPLAY SELECTION or MODIFY SELECTION.

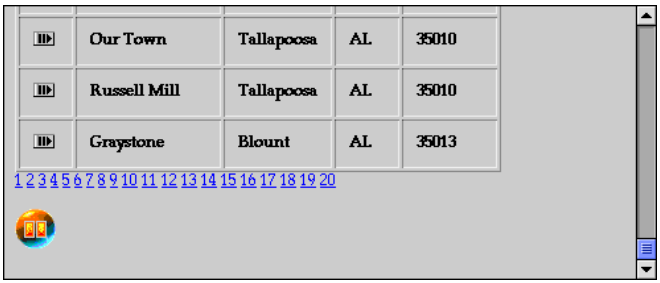
SET WEB DISPLAY LIMITS enables you to change these settings. In the numberRecords parameter, you indicate the maximum number of records you want to display per selection page. In numberPages, you indicate the maximum number of selection page links you want at the end of each selection page.

For example, if you have a selection of 10,000 records and want to browse all of them in one display selection, you can pass numberRecords=100 and numberPages=100. However, remember that the data is going over the network or Internet; with the Internet, you must take the speed factor into account when changing the display selection settings.

In addition, SET WEB DISPLAY LIMITS optionally allows you to change the default icon of the full page record button. In the picRef parameter, specify the picture reference number of the picture stored in the database Picture library you want to use as new icon.
SET WEB DISPLAY LIMITS only affects subsequent calls to DISPLAY SELECTION or MODIFY SELECTION, and its scope is local to the current process.

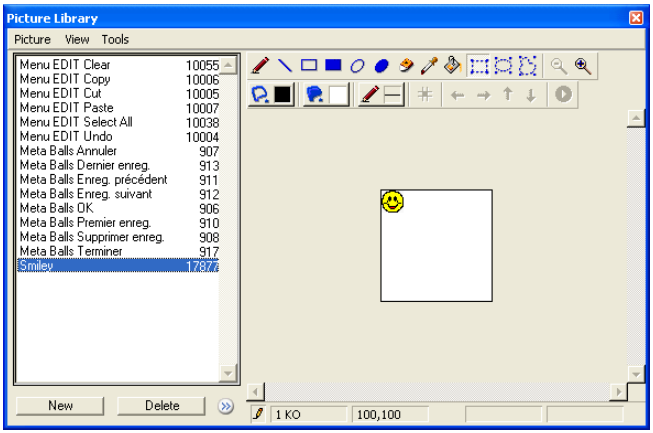
Example

In the following example, a DISPLAY SELECTION or a MODIFY SELECTION is issued for a [Zip Codes] table. By default, 4D displays the records on the Web browser side as shown here:



Note that the first 400 records can be browsed.

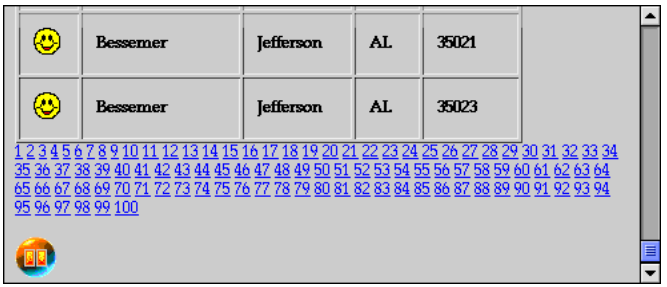
If the following picture is added to the database Picture Library:



And, if the project method that displays the selection performs the SET WEB DISPLAY LIMITS call shown here, prior to the call to DISPLAY SELECTION or MODIFY SELECTION:

```
SET WEB DISPLAY LIMITS (50;100;17877)
```

Then the selection on the Web browser side ends up looking like this:



You can now browse the first 5,000 records of the selection.

See Also

DISPLAY SELECTION, MODIFY SELECTION, Using the Contextual Mode.

SET WEB TIMEOUT

SET WEB TIMEOUT (timeout)

Parameter	Type		Description
timeout	Number	→	Web connections timeout expressed in seconds

Description

The SET WEB TIMEOUT command sets the timeout for the Web Connection processes in contextual mode. The default timeout is 5 minutes.

You reduce or increase this delay by passing, in the timeout parameter, the new timeout expressed in seconds.

The command takes effect immediately, and its scope is the working session.
• If SET WEB TIMEOUT is called from within a Web process, the value of timeout is applied to that process only.
• If SET WEB TIMEOUT is not called from a Web process, all the Web Connection processes are affected.

See Also

Using the Contextual Mode, Web Server Settings.

START WEB SERVER

Parameter	Type	Description

This command does not require any parameters

Description

The START WEB SERVER command starts the Web server of the 4D application on which it has been executed (4D Developer, 4D Server or 4D Client). 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.

See Also

STOP WEB SERVER.

System Variables and Sets

If the Web Server is successfully started, OK is set to 1; otherwise OK is set to 0.

STOP WEB SERVER

version 2003 (Modified)

STOP WEB SERVER

Parameter	Type	Description

This command does not require any parameters

Description

The STOP WEB SERVER command stops the Web server of the 4D application on which it has been executed (4D Developer, 4D Server or 4D Client). 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.

See Also

START WEB SERVER.

Validate Digest Web Password (userName; password) → Boolean

Parameter	Type		Description
userName	Text	→	User name
password	Text	→	User password
Function result	Boolean	←	True=Authentication OK, False=Authentication failed

Description

The Validate Digest Web Password command can be used to check 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)
        $0:=Validate Digest Web Password($user;[WebUsers]password)
    Else
        $0:=False `User does not exist
    End if
End if
```

See Also

On Web Authentication Database Method.

WEB CACHE 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	Number	←	Percentage of the cache used

Description

The WEB CACHE STATISTICS command allows you to obtain 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, please refer to section Web Server Settings.

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", you place the tag <!--4DACTION/STATS—>. Then you insert two 4D variables, *vPages* and *vUsage*.
In the project method STATS, you write the following code:

```
C_TEXT ($1)
ARRAY TEXT (pages;0)
ARRAY LONGINT (hits;0)
C_LONGINT (vUsage)
```

```
WEB CACHE STATISTICS(pages;hits;vUsage)
vPages:=Char(1)
For ($i;1;Size of array(pages))
        ` For each page present in the cache
    vPages:=vPages+pages{$i}+"   "+String(hits{$i})+"<br>"
        ` Insert the name of the page and the HTML code
End for
SEND HTML FILE("stats.shtm")
    ` The contents of the pages with the suffix ".shtm" is always parsed
```

See Also

Information about the Web Site, Web Server Settings.

Web Context → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	True = Contextual mode False = Non contextual mode

Description

The Web Context command must be called from a Web process. It returns a boolean that indicates if the Web connection is executing in contextual mode (True) or in non-contextual mode (False).

Note : The Web Context command always returns False when it is:
• called from a process other than a Web process,
• executed on a 4D Client machine.

The use of this function is advocated in the On Web Connection Database Method.

Example

Here is an example of the On Web Connection database method:

```
If (Web Context)
    WithContext ($1;$2;$3;$4;$5;$6)
Else
    WithoutContext ($1;$2;$3;$4;$5;$6)
End if
```

See Also

On Web Connection Database Method, PROCESS PROPERTIES, Using the Contextual Mode.

63

Web Services (Client)

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.

AUTHENTICATE WEB SERVICE (name; password)

Parameter	Type		Description
name	String	→	User name
password	String	→	User password

Description

The AUTHENTICATE WEB SERVICE command enables the use of Web Services requiring authentication of the client application (simple authentication).

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 CALL WEB SERVICE command. It is thus necessary to call the AUTHENTICATE WEB SERVICE command before calling the CALL WEB SERVICE command.

The authentication information is reset to zero after each request. Therefore, you must use the AUTHENTICATE WEB SERVICE command before each CALL WEB SERVICE command.

If authentication fails, the SOAP server returns an error that you can identify using the Get Web Service error info command.

See Also

CALL WEB SERVICE, Get Web Service error info.

CALL WEB SERVICE (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)
*	*	→	Do not close connection

Description

The CALL WEB SERVICE command is used to call a Web Service by sending an HTTP request. This request contains the SOAP message created previously using the SET WEB SERVICE PARAMETER command.

Any subsequent call to the SET WEB SERVICE PARAMETER command will cause the creation of a new request. The execution of the CALL WEB SERVICE 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.

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 SET WEB SERVICE PARAMETER and GET WEB SERVICE 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 Web Services (Client) theme:

Constant	Type	Value
Web Service Dynamic	Longint	0 (default)
Web Service Manual In	Longint	1
Web Service Manual Out	Longint	2
Web Service Manual	Longint	3

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 GET WEB SERVICE 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 SET WEB SERVICE 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)

SET WEB SERVICE PARAMETER("MyXMLBlob";$1)
CALL WEB SERVICE("http://my.domain.com/my_service";"MySoapAction";
                 "TheMethod";"http://my.namespace.com/";Web Service Manual In)
GET WEB SERVICE 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 GET WEB SERVICE 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)

SET WEB SERVICE PARAMETER("MyInputVar";$1)
CALL WEB SERVICE("http://my.domain.com/my_service";"MySoapAction";
                 "TheMethod";"http://my.namespace.com/";Web Service Manual Out)
GET WEB SERVICE 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)

SET WEB SERVICE PARAMETER("MyXMLInputBlob";$1)
CALL WEB SERVICE("http://my.domain.com/my_service";"MySoapAction";
                 "TheMethod";"http://my.namespace.com/";Web Service Manual)
GET WEB SERVICE 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)

SET WEB SERVICE PARAMETER("MyXMLInput";$1)
CALL WEB SERVICE("http://my.domain.com/my_service";"MySoapAction";
                 "TheMethod";"http://my.namespace.com/";Web Service Manual)
GET WEB SERVICE 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 CALL WEB SERVICE will reuse this same connection if the * parameter is passed, and so on. To close the connection, simply execute the CALL WEB SERVICE 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 CALL WEB SERVICE 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.

See Also

GET WEB SERVICE RESULT, SET WEB SERVICE PARAMETER.

System Variables or 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.

Get Web Service error info (infoType) → String

Parameter	Type		Description
infoType	Longint	→	Information to be retrieved
Function result	String	←	Information about the last SOAP error

Description

The Get Web Service error info command returns information about the last error encountered during the execution of a 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
Web Service Error Code	Longint	0
Web Service Detailed Message	Longint	1
Web Service HTTP Error code	Longint	2
Web Service Fault Actor	Longint	3

These constants are used to retrieve the following values:

• Web Service Error Code: Main error code (defined by 4D). This code is also returned in the *Error* system variable.
List of codes that may be returned:
9910: Soap fault (see also Web Service Fault Actor)
9911: Parser fault
9912: HTTP fault (see also Web Service HTTP Error code)
9913: Network fault
9914: Internal fault.

• Web Service Detailed Message: Detailed message describing the error. The type of message differs according to the main error type.
- If the main error = 9910 (Soap fault): the cause of the SOAP fault is returned (e.g.: “the remote method does not exist”).
- If the main error = 9911 (Parser fault): the location of the error in the XML document is returned.

- If the main error = 9912 (HTTP fault):
    - 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..
    - for any other HTTP error code, the <body> is returned.
- If the main error = 9913 (Network fault): the cause of the network fault is returned (e.g.: “ServerAddress: DNS lookup failure”)
- If the main error = 9914 (Internal fault): the cause of the internal fault is returned.

• Web Service HTTP Error code: HTTP error code (to be used in case of main error 9912).

• Web Service Fault Actor: Cause of the error (returned by the SOAP protocol — to be used in the case of main error 9910).
    - Version Mismatch
    - Must Understand (the server was unable to interpret a parameter defined as mandatory)
    - Client Fault
    - Server Fault
    - Encoding Unknown .

An empty string is returned when no information is available.

GET WEB SERVICE 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 GET WEB SERVICE RESULT command is used to retrieve a value sent back by the Web Service as a result of the processing performed.

Note: This command must be used only after the CALL WEB SERVICE 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 is used to specify 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)
```

```
SET WEB SERVICE PARAMETER("city";$1)
SET WEB SERVICE PARAMETER("country_code";$2)

CALL WEB SERVICE("http://www.citiesoftheworld.com/WS";"WSTime#City_time";
                 "City_time";"http://www.citiesoftheworld.com/namespace/default")

If (OK=1)
   GET WEB SERVICE RESULT($0;"return";*)
End if
```

See Also

CALL WEB SERVICE, SET WEB SERVICE PARAMETER.

SET WEB SERVICE 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 SET WEB SERVICE OPTION command allows you to set different options that will be used during the next SOAP request triggered using the CALL WEB SERVICE 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 (*option* param)	Type	Value
Web Service HTTP Timeout	Longint	1
Web Service SOAP Header	Longint	2
Web Service SOAP Version	Longint	3

Constant (*value* param)	Type	Value
Web Service SOAP_1_1	Longint	0
Web Service SOAP_1_2	Longint	1

The following details each of the options and possible values:

• option = Web Service HTTP Timeout
value = timeout of the client portion expressed in seconds.
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.
This timeout is 10 seconds by default. It can be modified for specific reasons (network status, Web Service specifics, etc.).

• option = Web Service SOAP Header
value = XML root element reference to insert as a header in the SOAP request.
This option allows you to insert a header in a SOAP request generated using the CALL WEB SERVICE command. SOAP requests do not contain a specific header by default. However, certain Web Services require a header, for example when managing identification parameters.

• option = Web Service SOAP Version
value = Web Service SOAP_1_1 or Web Service SOAP_1_2
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.

Examples

1. Insertion of a customized header in the SOAP request:

```
    ` Creating an XML reference
C_STRING(16;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
SET WEB SERVICE OPTION(Web Service SOAP Header;vElemRef)
```

2. Using version 1.2 of the SOAP protocol:

```
SET WEB SERVICE OPTION(Web Service SOAP Version;Web Service SOAP_1_2)
```

See Also

CALL WEB SERVICE.

SET WEB SERVICE 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 SET WEB SERVICE 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 "Compiler" and "Arrays" themes.

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)
SET WEB SERVICE PARAMETER("city";$1)
SET WEB SERVICE PARAMETER("country";$2)
```

See Also

CALL WEB SERVICE, GET WEB SERVICE RESULT.

64

Web Services (Server)

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.

Get SOAP info (infoNum) → String

Parameter	Type		Description
infoNum	Longint	→	Number of type of SOAP info to get
Function result	String	←	SOAP Information

Description

The Get SOAP info command is used to retrieve, in the form of a character string, the different types of information concerning a SOAP request.

When you process a SOAP request, it may 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
SOAP Method Name	Longint	1
SOAP Service Name	Longint	2

- SOAP Method Name = name of the Web Service method about to be executed.
- SOAP Service Name = 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 ("Structure Access" theme).

See Also

SEND SOAP FAULT, SET DATABASE PARAMETER.

Is SOAP request

Is SOAP request → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	True if the request is SOAP; otherwise, False

Description

The Is 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.

See Also

On Web Authentication Database Method, SOAP DECLARATION.

SEND SOAP 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 SEND SOAP FAULT command is used to return an error to a SOAP client indicating the origin of the fault: client or server. Using this command enables you to 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 in order 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")
```

See Also

Get SOAP info, SOAP DECLARATION.

SOAP DECLARATION

version 2004.3 (Modified)

SOAP DECLARATION (variable; type; input_output{; alias})

Parameter	Type		Description
variable	4D 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 is used to explicitly declare 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 are used to 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.
• Next, it can be useful or necessary to customize the parameter names (incoming and outgoing).
• Also, returning values with a size greater than 32 KB (limit for Text arguments) can be necessary.
• Finally, this operation makes it impossible to return more than one value per RPC call (in $0).

The SOAP DECLARATION command allows you to be free from these limits. You can execute this command for each incoming and outgoing parameter and assign it a name and a type.

Note: Even if the SOAP DECLARATION command is used, it is always 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. 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:

Constant	Type	Value
Is BLOB	Longint	30
Is Boolean	Longint	6
Is Integer	Longint	8
Is LongInt	Longint	9
Is Real	Longint	1
Boolean array	Longint	22
String array	Longint	21
Date array	Longint	17
Integer array	Longint	15
LongInt array	Longint	16
Real array	Longint	14
Text array	Longint	18
Is Text	Longint	2
Is Date	Longint	4
Is Time	Longint	11
Is String Var	Longint	24

Note: The following constants are not used in SOAP methods: Is Alpha Field, Is Pointer, Array 2D, Picture array, Pointer array, Is Picture, Is Subtable, Is Undefined.

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

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 Web Services, Special 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.

Examples

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")
```

2. This example is used to retrieve 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")
```

3. This example is used to refer to two return values without specifying an argument name:

```
SOAP DECLARATION(ret1;Is LongInt;SOAP Output)
SOAP DECLARATION(ret2;Is LongInt;SOAP Output)
```

4. This example allows the 4D SOAP server to return an argument with a size greater than 32 KB:

```
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.

See Also

Get SOAP info, Is data file locked, SEND SOAP FAULT.

Constants

Field and Variable Types and Web Services (Server) themes.

65

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.

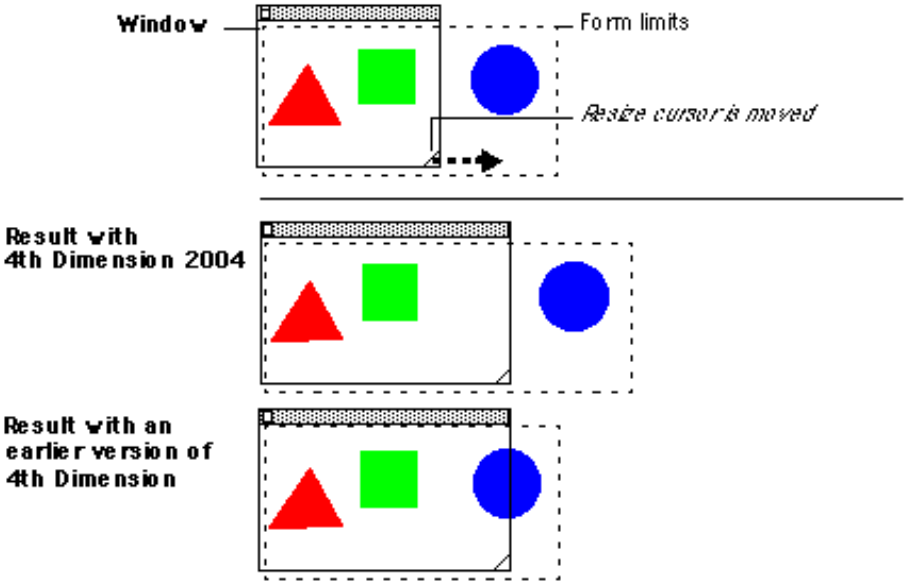
You can open various types of custom windows with the Open Window or Open form window commands (see the Window Types section). All windows opened by these commands are referenced through a **WinRef** expression. A WinRef is the unique ID of each open window. It is a Longint expression. All commands working 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 GRAPH TABLE, 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
Starting with version 2004 of 4D, 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 SET FORM 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
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.

See Also

Open form window, Open window, 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	Can be a floating window
Plain window	Long Integer	8	No
Plain no zoom box window	Long Integer	0	No
Plain fixed size window	Long Integer	4	No
Modal dialog box	Long Integer	1	No
Alternate dialog box	Long Integer	3	Yes
Movable dialog box	Long Integer	5	Yes
Plain dialog box	Long Integer	2	Yes
Palette window	Long Integer	1984	Yes
Round corner window	Long Integer	16	No
Pop up window	Long Integer	32	No
Sheet window	Long Integer	33	No
Resizable sheet window	Long Integer	34	No

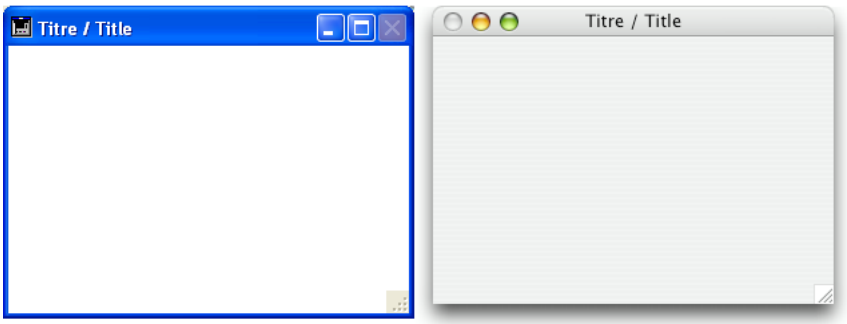
Floating Windows: If you pass one of these constants to Open window, you open a regular windows. To open a floating windows, pass a negative window type value to Open window.

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.

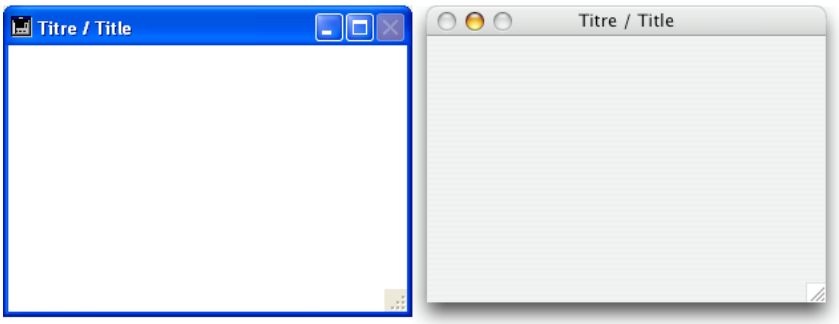
The following table shows each window type, on Windows (left) and on Macintosh (right).

Plain window (8)



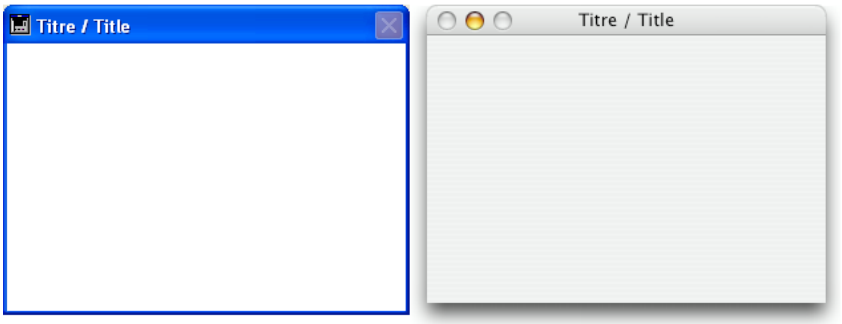
- Can have a title: Yes
- Can have a close box or equivalent: Yes
- Can be resized: Yes
- Can be minimized/maximized or zoomed: Yes
- Suitable for scroll bars: Yes
- Usage: data entry with scrollbars, DISPLAY SELECTION, MODIFY SELECTION, etc.

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 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

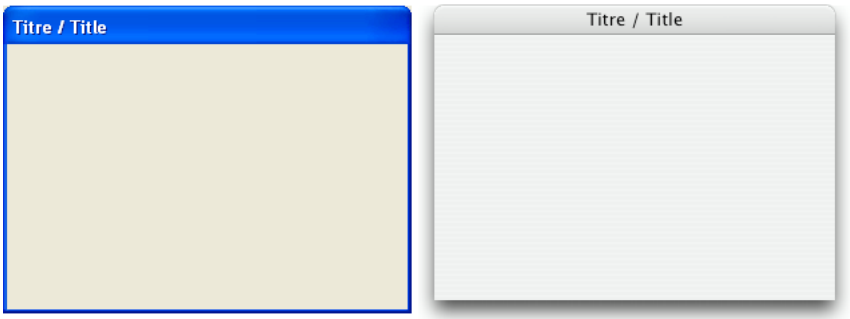
Modal dialog box (1)

- 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

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

Movable dialog box (5)

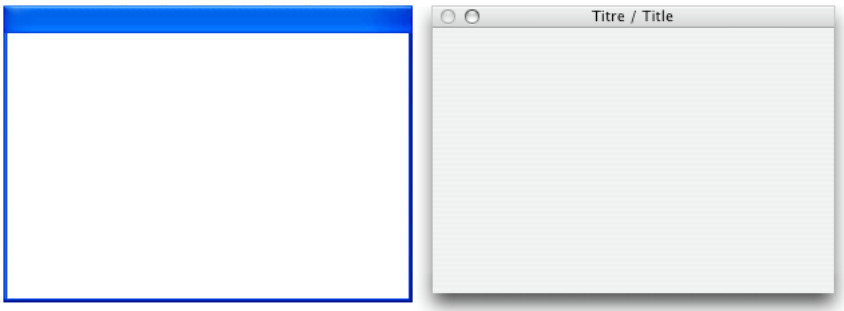


- Can have a title: Yes
- Can have a close box or equivalent: No
- Can be resized: No
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No
- Usage: DIALOG, ADD RECORD(...;...;*) or equivalent
- Windows of this type are modal, but can be moved and can be used as floating windows

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 {+ 1} {+ 2} {+ 4} {+ 8})

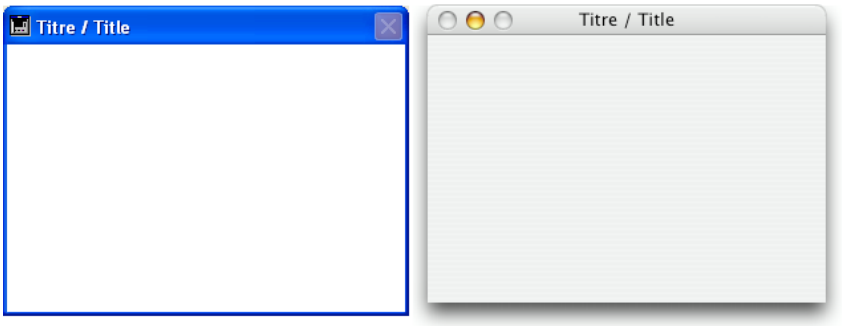


When you call Open window, you can add one or several of the following constants to Palette window in order to obtain variations in the behavior of the window:

Constant	Type	Value
Has zoom box	Long Integer	8
Has grow box	Long Integer	4
Has window title	Long Integer	2
Has highlight	Long Integer	1

- Can have a title: Yes, if Has window title variation is specified
- Can have a close box or equivalent: Yes
- Can be resized: Yes, if Has grow box variation is specified
- Can be minimized/maximized or zoomed: Yes, if Has zoom box variation is specified
- Suitable for scroll bars: Yes, if Has grow box variation is specified
- Usage: Floating windows with DIALOG or DISPLAY SELECTION (no data entry)

Round corner window (16)

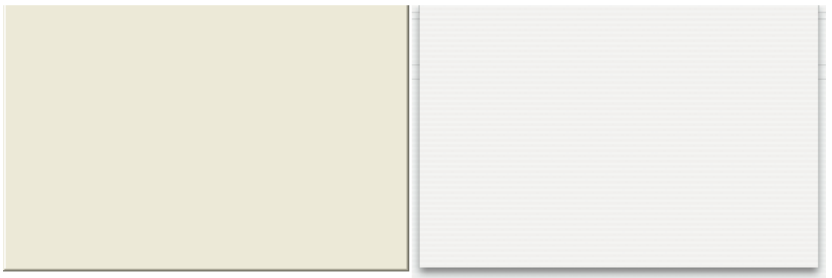


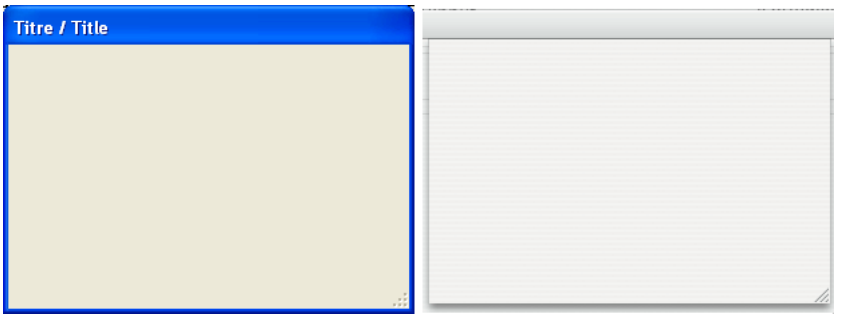
- Can have a title: Yes
- Can have a close box or equivalent: Yes
- Can be resized: No on Macintosh
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No on Macintosh
- Usage: Rare (obsolete)

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 handle pop-up menus related to 3D "bevel" or "toolbar" type buttons.

Sheet window (33) and **Resizable sheet window** (34)

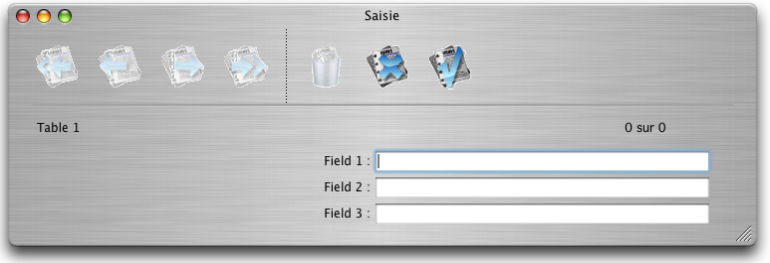




Sheet windows are specific to Mac OS X. These windows “drop down” over the title bar of the main window using animation and are displayed above the main window. They are automatically centered in the main window. Their properties are identical to those of the modal dialog boxes. They are generally used 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).

Metal Look (2048)



Under Mac OS, it is possible to apply the metal look to windows. This type of look is found throughout the Macintosh interface. Under Windows, this property has no effect.

To apply the metal look to a window created by the Open window command, you can just add the Metal Look constant to the window type set in the type parameter. For example:

```
$win:=Open window(10;80;-1;-1;Plain window+Metal Look;"")
```

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

See Also

Open external window, Open 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 extWindowRef parameter, CLOSE WINDOW closes the specified external window. For more information about external windows, refer to the Open external window function.

Example

The following example opens a window and adds new records with the ADD RECORD command. When the records have been added, the window is closed with CLOSE WINDOW:

```
Open window (5; 40; 250; 300; 0; "New Employee")
Repeat
   ADD RECORD ([Employees])  ` Add a new employee record
Until (OK = 0)  ` Loop until the user cancels
CLOSE WINDOW  ` Close the window
```

See Also

Open external window, Open form window, Open window.

Current form window

Current form window → WinRef

Parameter	Type		Description
This command does not require any parameters			
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.

See Also

Open form window, Open window, RESIZE FORM WINDOW.

DRAG WINDOW

Parameter	Type	Description

This command does not require any parameters

Description

The DRAG WINDOW command allows users to 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 frame created with a static picture, 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:

```
Open window(50;50;50+400;50+300;2)
DIALOG([Table1];"Custom Drag")
CLOSE WINDOW
```

You obtain a window similar to this:

Then you can drag the window by clicking anywhere on the borders.

See Also

GET WINDOW RECT, SET WINDOW RECT.

ERASE WINDOW

version 6.0 (Modified)

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.

See Also

GOTO XY, MESSAGE.

Find window (left; top{; windowPart}) → WinRef

Parameter	Type		Description
left	Number	→	Global left coordinate
top	Number	→	Global top coordinate
windowPart	Number	←	Window part ID number
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).

If you specify the windowPart parameter, whether or not a window has been found, the parameter returns one of the following values:

Constants	Type	Value	Platform
In menu bar	Long Integer	1	Macintosh only
In system window	Long Integer	2	Macintosh only
In contents	Long Integer	3	Windows or Macintosh
In drag	Long Integer	4	Macintosh only
In grow	Long Integer	5	Macintosh only
In go away	Long Integer	6	Macintosh only
In zoom box	Long Integer	7	Macintosh only

See Also

Frontmost window, Next window.

Frontmost window

Frontmost window {(*)} → WinRef

Parameter	Type		Description
*	*	→	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.

See Also

Frontmost process, Next window.

GET WINDOW RECT (left; top; right; bottom{; window})

Parameter	Type		Description
left	Number	←	Left coordinate of window's contents area
top	Number	←	Top coordinate of window's contents area
right	Number	←	Right coordinate of window's contents area
bottom	Number	←	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 command WINDOW LIST.

See Also

SET WINDOW RECT.

Get window title

Get window title {(window)} → String

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 command SET WINDOW TITLE.

See Also

SET WINDOW TITLE.

HIDE WINDOW {(window)}

Parameter	Type		Description
window	WinRef	→	Window reference number or Current process frontmost window, if omitted

Description

The HIDE WINDOW command allows you to hide the window whose number was passed in window or, if this parameter is omitted, the current process frontmost window. For example, this command allows you to 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 programmatically apply any changes supported by 4D windows.

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
```

See Also

SHOW 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:

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

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

Notes:

• This command only applies to windows that contain a zoom box. If the window type does not include it, the command does nothing. For more information, please refer to the Window Types section.

• 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.
• If the window is already maximized, the command does nothing.

MAXIMIZE WINDOW sets a window 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.

A later click on the zoom box of the window 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.

Example

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
```

See Also

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

See Also

MAXIMIZE WINDOW.

Next window

Next window (window) → WinRef

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).

See Also

Frontmost window.

Open external window

Windows

version 6.0 (Modified)

Open external window (left; top; right; bottom; type; title; plugInArea) → Number

Parameter	Type		Description
left	Number	→	Global left coordinate of window contents area
top	Number	→	Global top coordinate of window contents area
right	Number	→	Global right coordinate of window contents area
bottom	Number	→	Global bottom coordinate of window contents area
type	Number	→	Window type
title	String	→	Title of window
plugInArea	String	→	External area command
Function result	Number	←	Reference number for window and external area

Description

Open external window opens a new window and displays the external area supported by the command plugInArea provided by a 4D plug-in. The code passed in plugInArea is generally in the form "_PluginName", for example: _4D Write, _4D View or _4D Draw.

Open external window returns a Long Integer value that can be used both as a window reference number (that can be used with other Windows theme commands) and as a reference to the external area displayed in the window (that can be used with other routines provided by the 4D plug-in).

The first six arguments are the same as those of the the Open window command. However, none of the parameters are optional.

Open external window creates modeless windows. The command does not wait for user input, so you can have several active windows open at once. You can click between each window and edit the one in front. If the window type has a title bar, a Control-menu box (Windows) or a Close Box (Macintosh) will be added to enable the user to close the window.

Examples

The following example opens an external window and displays the 4D Write external area:

```
wrWind:=Open external window (50; 50; 350; 450; 8; "Letter Writing"; "_4D WRITE")
```

The following example closes the external window opened in the previous example:

```
CLOSE WINDOW (wrWind)
```

See Also

CLOSE WINDOW, Open window.

Open form window

version 11 (Modified)

Open form window ({aTable; }formName{; type{; hPos{; vPos{; *}}}}) → WinRef

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
*	*	→	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 GET FORM 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
Standard form window	Longint	8
Modal form dialog box	Longint	1
Movable form dialog box	Longint	5
Palette form window	Longint	1984
Pop up form window	Longint	32
Sheet form window	Longint	33

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.
• 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.

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
Vertically Centered	Longint	262144
At the Top	Longint	327680
At the Bottom	Longint	393216

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.

Examples

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")
```

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;*)
```

See Also

GET FORM PROPERTIES, Open window, Window Types.

Open window (left; top; right; bottom{; type{; title{; controlMenuBox}}}){ → WinRef }

Parameter	Type		Description
left	Number	→	Global left coordinate of window contents area
top	Number	→	Global top coordinate of window contents area
right	Number	→	Global right coordinate of window contents area, or -1 for using form default size
bottom	Number	→	Global bottom coordinate of window contents area, or -1 for using form default size
type	Number	→	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.

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 command INPUT FORM, to which you passed the optional * parameter.

Important: This automatic sizing of the window will occur only if you made a prior call to INPUT FORM for the form to be displayed, and if you passed the * optional parameter to INPUT FORM.

• 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. 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 INPUT FORM for the form to be displayed, and if you passed the * optional parameter to INPUT FORM.

• 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.

Version 6 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.

Examples

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
```

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.

```
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
```

3. The following example opens a window whose size and title come from the properties of the form displayed in the window:

```
INPUT FORM([Customers];"Add Records";*)
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 INPUT FORM with the optional * parameter, and the properties of the form must have been set accordingly in the Design environment.

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
```

See Also

CLOSE WINDOW, Open external window, Open form 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.

See Also

ERASE 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 SET FORM 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:

See Also

GET FORM PROPERTIES, SET FORM SIZE, SET WINDOW RECT.

SET WINDOW RECT

Windows

version 2004 (Modified)

SET WINDOW RECT (left; top; right; bottom{; window})

Parameter	Type		Description
left	Number	→	Global left coordinate of window's contents area
top	Number	→	Global top coordinate of window's contents area
right	Number	→	Global right coordinate of window's contents area
bottom	Number	→	Global bottom coordinate of window's contents area
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted

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).

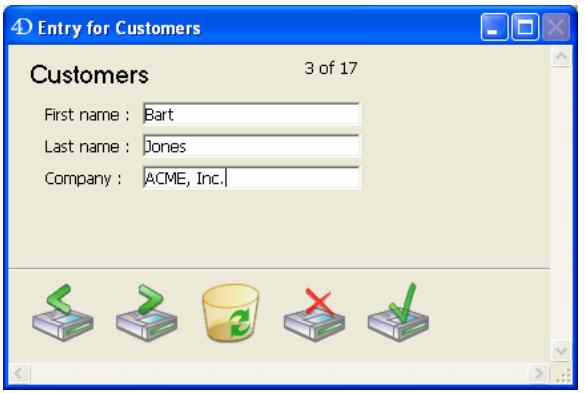
Warning: Be aware that by using this command, you may move a window beyond the limits of the main window (on Windows) or of the screens (on Macintosh). To prevent this, use commands such as Screen width and Screen height to double-check the new coordinates of the window.

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.

Examples

1. See example for the WINDOW LIST command.

2. Given the following window:



After execution of the following line:

```
SET WINDOW RECT(100;100;300;300)
```

The window appears as follows:

See Also

DRAG WINDOW, GET WINDOW RECT, RESIZE FORM WINDOW.

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
```

See Also

Get window title.

SHOW WINDOW {(window)}

Parameter	Type		Description
window	WinRef	→	Window reference number or Current process frontmost window, if omitted

Description

The SHOW WINDOW command allows you to display 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.

See Also

HIDE WINDOW.

Window kind

Window kind {(window)}

Parameter	Type		Description
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted

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 values:

Constant	Type	Value
Regular window	Long Integer	8
Modal dialog	Long Integer	9
External window	Long Integer	5
Floating window	Long Integer	14

If you omit the window parameter, Window kind returns the type of the frontmost window for the current process.

Example

Set example for the command WINDOW LIST.

See Also

GET WINDOW RECT, Get window title, Window process.

WINDOW LIST (windows{; *})

Parameter	Type		Description
windows	Array	←	Array of window reference numbers
*	*	→	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).

See Also

Window kind, Window process.

Window process

Window process {(window)} → Number

Parameter	Type		Description
window	WinRef	→	Window reference number
Function result	Number	←	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.

See Also

Current process.

66

XML

4D includes a set of commands used for parsing objects containing XML (*eXtensible Markup Language*) data.

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 commands and SAX commands

4D offers two separate sets of XML commands, prefixed DOM and SAX.

What are 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.
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 GET XML ERROR command (common to both XML standards).
The DOM CLOSE XML command lets you close the source in the end.

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.

Use of XPath notation (DOM)

Three 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>
```

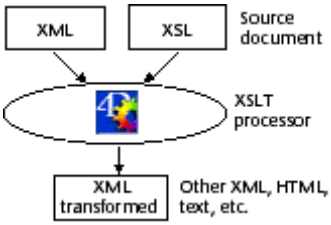
XPath notation allows you to access the “ccc” value using the /RootElement/Elem1/Elem2[3] syntax.

For an illustration of XPath notation, please refer to the examples in the DOM Create XML element and DOM Find XML element commands.

XSL transformations management

4D supports the application of XSL style sheets (eXtended Stylesheet Language). The XSL language allows you to modify the tags of an XML document.

The XSL language features two different aspects:
• **Formatting**: It allows you to apply style and display rules to XML elements, similar to CSS (Cascading StyleSheet) in HTML.
• **Transformations**: It allows you to change XML tags to a different tag system, for example, HTML. This function is known as **XSLT**. An XSL style sheet can totally reorganize the XML elements of a document by selecting them then transforming them into other elements. This function is useful, for example, for synchronizing a set of dissimilar documents.



Note: 4D uses the Xalan-C_1_6_0.dll library to perform XSL transformations. Xalan is a freeware XSLT processor. For more information, please visit http://xml.apache.org/xalan-c/index.html.

XSL style sheets are text documents (with .xsl extensions) generated manually or using specialized applications. The XSL language features various elements and functions that allow you to perform any type of dynamic transformation. For more information on this language, please visit http://xmlfr.org (for example).

4D allows you to transform an XML document using an existing XSL style sheet (APPLY XSLT TRANSFORMATION command). Also, 4D allows you to modify XSL style sheet parameters on the fly using the SET XSLT PARAMETER command.

Note: An option in the export dialog box lets you use an XSL style sheet when exporting XML and thus generate a transformed XML document.

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 it consists of 16 characters.

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.

APPLY XSLT 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)

Description

The APPLY XSLT TRANSFORMATION command applies an XSLT transformation to a document or a BLOB containing XML and generates a document or a BLOB result. The scope of this command is the current process.

Note: For more information about XSL transformation (or XSLT), refer to the Overview of XML Commands section.

The command requires three BLOBs or character string parameters. **Warning**: This command only accepts variables or fields as parameters.

If you pass a character string, you designate a document. In this case, you can only pass the name (the document must be next to the database structure) or the full access path of the document.
You cannot mix different types of parameters within the same call.

• The xmlSource parameter must contain the XML source to transform. The command checks the validity of the XML code.
• The xslSheet parameter must contain the XSL style sheet to use for the XSLT transformation. This style sheet may have been generated manually or using speciality software. The command checks the validity of the XML code.
• The result parameter must contain the name of the document or the BLOB that must receive the result of the XSLT transformation. If you pass a document name that does not exist at the designated location, 4D creates it automatically. If the document is already open with write access, an error is generated.

The command parses the XML source and transforms it using the instructions in the XSL style sheet. If the SET XSLT PARAMETER command was used beforehand, the command replaces the parameters defined by their value. The result of the transformation is written in the document or BLOB result.

The optional compileSheet parameter can be used to optimize the XSLT transformation, more particularly in the case of successive applications of the same XSL style sheet. When the compileSheet parameter is passed and is set to True, the XSL file xslSheet is parsed on the first call of the command, then is compiled and stored in memory. On each subsequent call with the same XSL sheet, the command uses the compiled file directly (unless it has been modified), which can accelerate processing. Optimization does not take into account any modifications carried out in the imported file (via xsl:import). If a file referenced by the XSL file is modified, it is necessary to "force" the recompiling of a new XSL file by calling the command again with the compileSheet parameter set to False (or omitted).

Example

Refer to the example of the SET XSLT PARAMETER command.

See Also

GET XSLT ERROR, SET XSLT PARAMETER.

System Variables or Sets

If the transformation was executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

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.

See Also

DOM Parse XML source, DOM Parse XML variable.

System Variables or 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

version 2004 (Modified)

DOM Count XML attributes (elementRef) → Longint

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 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_STRING(16;$xml_Parent_Ref;$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;$numAttributes)
For($i;1;$numAttributes)
    DOM GET XML ATTRIBUTE BY INDEX($xml_Child_Ref;$i;tAttrib{$i})
End for
```

In the above example, $numAttributes equals 3, tAttrib{1} contains “Font,” tAttrib{2} contains “N” and tAttrib{3} contains “size.”

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).

See Also

DOM Count XML elements.

System Variables or 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

version 2004 (Modified)

DOM Count XML elements (elementRef; elementName) → Longint

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.

See Also

DOM Get XML element.

System Variables or 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.

version 11 (Modified)

DOM Create XML element (elementRef; xPath{; attrName; attrValue}{; attrName2; attrValue2; ...; attrNameN; attrValueN}) → String

Parameter	Type		Description
elementRef	String	→	Root XML element reference
xPath	Text	→	XPath path of the XML element to create
attrName	String	→	Attribute to set
attrValue	String | Value	→	New attribute value
Function result	String	←	Reference of the created XML element

Description

The DOM Create XML element command allows you to create a new element in the XML element elementRef in the path set by the xPath parameter and to add 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 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" (not translated)
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.

Examples

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_STRING(16;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)
```

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_STRING(16;vRootRef;vElemRef)
C_STRING(80;$aAttrName1;$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)
```

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_STRING(16;$root)
C_STRING(16;$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)
```

See Also

DOM Get XML element, DOM REMOVE XML ELEMENT.

System Variables or 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 Handling

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 Ref

version 2004

DOM Create XML Ref (root{; nameSpace}{; nameSpaceName; nameSpaceValue}{; nameSpaceName2; nameSpaceValue2; ...; nameSpaceNameN; nameSpaceValueN}) → String

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”).
In this case, you must prefix the root parameter with the namespace name followed by **:** (for example, “MyNameSpace:MyRoot”).

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.

Examples

1. Creating a single XML tree:

```
C_STRING (16;vElemRef)
vElemRef:=DOM Create XML Ref("MyRoot")
```

This code produces the following result:
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<MyRoot/>

2. Creating an XML tree with a single namespace:

```
C_STRING (16;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"/>

3. Creating an XML tree with several namespaces:

```
C_STRING (16;vElemRef)
C_STRING (80;$aNSName1;$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"/>

See Also

DOM CLOSE XML, DOM SET XML OPTIONS.

System Variables or 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

version 2004

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 allows you to store 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.

Example

This example stores the tree vElemRef in the file MyDoc.xml:

```
DOM EXPORT TO FILE(vElemRef;"C:\\folder\MyDoc.xml")
```

See Also

DOM EXPORT TO PICTURE, DOM EXPORT TO VAR.

System Variables or 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 Handling

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 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 (default) = Own data source

Description

The DOM EXPORT TO PICTURE command can be used to save an SVG format picture contained in an XML tree in the picture field or variable indicated by the pictVar parameter.

SVG (Scalable Vector Graphics) is a file format used to describe vector graphics (extension .svg) in XML. 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 most common use of SVG is the publication of statistical or mapping data. For more information about this format, please refer to the following address: http://www.w3.org/Graphics/SVG/.

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 can be used to define 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:
•Get XML Data Source (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.
• Copy XML Data Source (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.
• Own XML Data Source (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 ELEMENT VALUE($ref;"Hello World")
DOM EXPORT TO PICTURE($svg;vpict;Copy XML Data Source)
DOM CLOSE XML($svg)
```

See Also

DOM EXPORT TO FILE, 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 allows you to save 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.

Example

This example stores the tree vElemRef in a text variable:

```
C_TEXT(vtMyText)
DOM EXPORT TO VAR(vElemRef;vtMyText)
```

See Also

DOM EXPORT TO FILE, DOM EXPORT TO PICTURE.

System Variables or 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).

version 11 (Modified)

DOM Find XML element (elementRef; xPath{; arrElementRefs}) → elementRef

Parameter	Type		Description
elementRef	String	→	XML element reference
xPath	Text | BLOB	→	XPath path of the element to look for
arrElementRefs	String 16 array	←	List of element references found (if applicable)
Function result	elementRef	←	Reference of the element found (if applicable)

Description

The DOM Find XML element command allows you to look 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 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.

Examples

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.

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 STRING(16;arrAfound;0)
vFound:=DOM Find XML element(vElemRef;"/Root/Elem1/Elem2";arrAfound)
```

See Also

DOM Count XML elements, DOM Create XML element.

System Variables or 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 Handling

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) → String

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 can be used to search for an XML element based on the value of its ID attribute. The search begins at the element designated by the elementRef parameter.

The ID attribute can be used to associate a unique identifier to each element. The value of the ID attribute must be a valid XML name and must be unique. Pass the value of the attribute to be searched for in id.

The command returns the XML reference of the element found as a result.

See Also

DOM Find XML element.

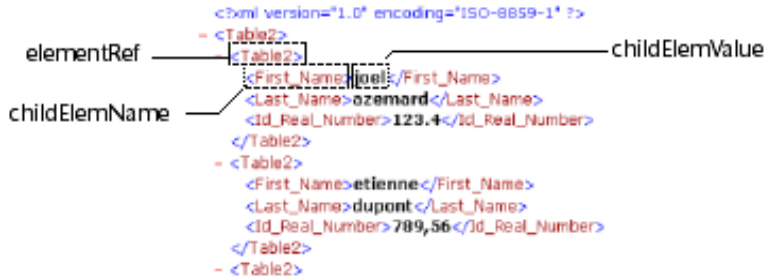
DOM Get first child XML element (elementRef{; childElemName{; childElemValue}}) → String

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.



Examples

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_STRING(16;$xml_Parent_Ref;$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)
```

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_STRING(16;$xml_Parent_Ref;$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)
```

See Also

DOM Get next sibling XML element.

System Variables or 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

version 2004.2

DOM Get last child XML element (elementRef{; childElemName{; childElemValue}}) → String

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_STRING(16;$ref_XML_Parent;$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)
```

See Also

DOM Get first child XML element.

System Variables or 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

version 2004.2 (Modified)

DOM Get next sibling XML element (elementRef{; siblingElemName{; siblingElemValue}}) → String

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.

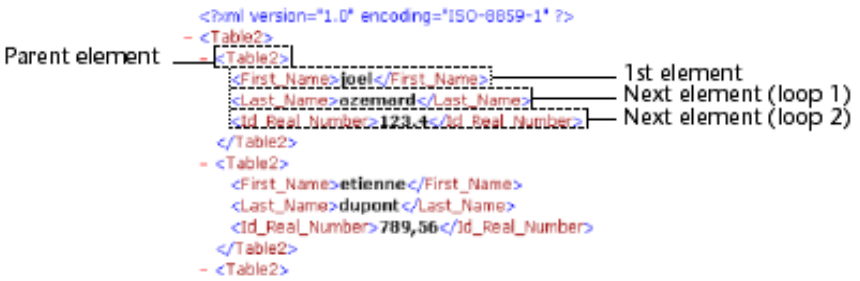
Examples

1. Retrieval of the reference of the next sibling XML element following the element passed as parameter:

```
C_STRING(16;$xml_Parent_Ref;$next_XML_Ref)
$next_XML_Ref:=DOM Get next sibling XML element($xml_Parent_Ref)
```

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_STRING(16;$xml_Parent_Ref;$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
```



See Also

DOM Get first child XML element.

System Variables or 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

version 2004.2

DOM Get parent XML element (elementRef{; parentElemName{; parentElemValue}}) → String

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.

See Also

DOM Get first child XML element, DOM Get last child XML element.

System Variables or 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

version 2004.2

DOM Get previous sibling XML element (elementRef{; siblingElemName{; siblingElemValue}}) → String

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.

See Also

DOM Get next sibling XML element.

System Variables or 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 XML ATTRIBUTE BY INDEX

version 2004 (Modified)

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 is used to get 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 will attempt 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.

See Also

DOM GET XML ATTRIBUTE BY NAME.

System Variables or 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 (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 is used to get 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 will attempt 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_STRING(16;$xml_Parent_Ref;$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>
```

See Also

DOM GET XML ATTRIBUTE BY INDEX.

System Variables or 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 (elementRef; elementName; index; elementValue) → String

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.

See Also

DOM GET XML ELEMENT VALUE.

System Variables or 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

version 2004 (Modified)

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 Commands section.

Example

This method returns the name of the $xml_Element_Ref element:

```
C_STRING(16;$xml_Element_Ref)
C_TEXT($name)

DOM GET XML ELEMENT NAME($xml_Element_Ref;$name)
```

See Also

DOM Get XML element, DOM GET XML ELEMENT VALUE, DOM SET XML ELEMENT NAME.

System Variables or 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.

version 2004 (Modified)

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.

Example

This method returns the value of the $xml_Element_Ref element:

```
C_STRING(16;$xml_Element_Ref)
C_REAL($value)

DOM GET XML ELEMENT VALUE($xml_Element_Ref;$value)
```

See Also

DOM Get XML element, DOM GET XML ELEMENT NAME, DOM SET XML ELEMENT VALUE.

System Variables or 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

version 2004 (Modified)

DOM Get XML information (elementRef; xmlInfo) → String

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 is used to retrieve 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
PUBLIC ID	Longint	1
SYSTEM ID	Longint	2
DOCTYPE Name	Longint	3
Encoding	Longint	4
Version	Longint	5
Document URI	Longint	6

These constants indicate the following information:
• *PUBLIC ID*: Public identifier (FPI) of the DTD to which the document conforms (if the DOCTYPE xxx PUBLIC tag is present).
• *SYSTEM ID*: System identifier.
• *DOCTYPE Name*: Name of the root element as defined in the DOCTYPE marker.
• *Encoding*: Encoding used (UTF-8, ISO...).
• *Version*: Accepted XML version.
• *Document URI*: URL of the DTD.

See Also

GET XML ERROR.

Constants

XML theme.

DOM Parse XML source (document{; validation{; dtd | schema}}) → String

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.

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 is used to indicate whether or not to validate the structure.
• If validation equals True, the structure will be validated. In this case, the parser will attempt 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 will not be validated.
If you pass True in validation and omit the third parameter, the command will attempt 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 will attempt to carry out both validations.

The third parameter is used to indicate a specific DTD or an XML schema for document parsing. If you use this parameter, the command will 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.

Examples

1. Opening an XML document located on disk, without validation:

```
$xml_Struct_Ref:=DOM Parse XML source("C:\\import.xml")
```

2. Opening an XML document located next to the database structure file, without validation:

```
$xml_Struct_Ref:=DOM Parse XML source("import.xml")
```

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")
```

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")
```

See Also

DOM CLOSE XML, DOM Parse XML variable.

System Variables or 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

version 2004 (Modified)

DOM Parse XML variable (variable{; validation{; dtd}}) → String

Parameter	Type		Description
variable	BLOB/Text	→	Name of the variable
validation	Boolean	→	True = Validation by the DTD, False = No validation
dtd	String	→	Location of the DTD
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 document.

Pass the name of the BLOB or Text variable containing the XML object in the variable parameter.

The Boolean parameter validation is used to indicate whether or not to validate the structure using the DTD.
• If validation equals True, the structure will be validated. In this case, the parser will attempt to validate the XML structure of the document based either on the DTD defined or referred to in the document, or that designated by the dtd parameter.
• If validation equals False, the structure will not be validated.

The third parameter, dtd, is used to indicate the specific DTD for document parsing. If you use this parameter, the command will not take the DTD referred to in the XML variable into account.

There are two ways to specify a DTD:
• as a *reference*. To do this, pass the complete pathname of the new DTD 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 *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.

If validation cannot be performed (no DTD, incorrect URL to DTD, 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.

Examples

1. Opening an XML object located in a 4D Text variable, without validation:

```
C_TEXT(myTextVar)
C_TIME(vDoc)
C_STRING(16;$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
```

2. Opening an XML document located in a 4D BLOB, without validation:

```
C_BLOB(myBlobVar)
C_STRING(16;$ref_XML_Struct)

DOCUMENT TO BLOB("c:\\import.xml";myBlobVar)
$xml_Struct_Ref:=DOM Parse XML variable(myBlobVar)
```

See Also

DOM CLOSE XML, DOM Parse XML source.

System Variables or 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 ELEMENT

version 2004

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.

See Also

DOM Create XML element.

System Variables or 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,
• The element is empty.

version 11 (Modified)

DOM SET XML ATTRIBUTE (elementRef; attrName; attrValue{; attrName2; attrValue2; ...; attrNameN; attrValueN})

Parameter	Type		Description
elementRef	String	→	XML element reference
attrName	String	→	Attribute to set
attrValue	String | Value	→	New attribute value

Description

The DOM SET XML ATTRIBUTE command allows adding one or more attributes to the XML element whose reference is passed in the elementRef parameter. It also allows setting the value of each attribute defined.

Pass the attriibute 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" (not translated)
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>
```

See Also

DOM GET XML ATTRIBUTE BY INDEX, DOM GET XML ATTRIBUTE BY NAME.

System Variables or 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 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 allows you to modify 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 will also take 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>
```

See Also

DOM GET XML ELEMENT NAME.

System Variables or 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 Handling

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).

version 2004.4 (Modified)

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
*	*	→	If passed: set the value in CDATA

Description

The DOM SET XML ELEMENT VALUE command allows you to modify 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 "Use of XPath notation" paragraph in the Overview of XML Commands section). 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 will be used "as is" in the XML structure.
• If you pass a variable or a field, 4D will process 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).

Examples

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>
```

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 &lt; 18</Postulate>
</Maths>
```

```
        ` CDATA form
        DOM SET XML ELEMENT VALUE(vElemRef;"12<18";*)
```

We get:

```
<Maths>
        <Postulate><![CDATA[12 < 18]]></Postulate>
</Maths>
```

See Also

DOM GET XML ELEMENT VALUE.

System Variables or 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 SET XML OPTIONS (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	→	True (default) = document is indented False = document is not indented

Description

The DOM SET XML OPTIONS command allows you to define various options that are useful in creating the XML tree set using elementRef. These options concern encoding, the standalone property and tree identation options:

• 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.

• 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.

• indentation: Indicates whether the tree should display indentations corresponding to XML key hierarchies (True) or not (False). By default (if the command is not called or if the parameter is omitted), the tree is indented.

Example

The following example sets the encoding to use and the standalone option in the elementRef element:

```
DOM SET XML OPTIONS(elementRef;"UTF-16";True)
```

See Also

DOM Create XML Ref.

GET XML ERROR

version 2003

GET XML 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 GET XML 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.

See Also

DOM Get XML information.

System Variables or 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.

GET XSLT ERROR (errorText{; row{; column}})

Parameter	Type		Description
errorText	Variable	←	Text of the error
row	Variable	←	Row number
column	Variable	←	Column number

Description

The GET XSLT ERROR command returns, in the errorText parameter, a description of the last error encountered during the XSLT transformation performed in the current process. The information returned is supplied by the Xerces.dll library.

The optional row and column parameters indicate the location of the error in the XSL document: 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.

See Also

APPLY XSLT TRANSFORMATION, SET XSLT PARAMETER.

System Variables or Sets

If the command has been executed correctly, the system variable OK is set to 1. If an error occurs, it is set to 0.

SAX ADD PROCESSING INSTRUCTION (document; statement)

Parameter	Type		Description
document	DocRef	→	Reference of open document
statement	Text	→	Text or BLOB 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(Quotes)+"text/xsl"+Char (Quotes)+ "href="+
                                                    Char (Quotes)+"style.xsl"+Char (Quotes)
SAX ADD PROCESSING INSTRUCTION ($DocRef;vtInstruct)
```

... will write the following line in the document:

```
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
```

See Also

SAX GET XML PROCESSING INSTRUCTION.

System Variables or 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 (document; data)

Parameter	Type		Description
document	DocRef	→	Reference of open document
data	Text | BLOB	→	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 </CData> 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 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 CDATAL($DocRef;vtMytext)
```

The result will thus be:

```
<![CDATA[
function matchwo(a,b)
{
if (a < b && a < 0) then
        {
        return 1
        }
else
        {
        return 0
        }
}
]]>
```

See Also

SAX GET XML CDATA.

System Variables or 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 (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-->
```

See Also

SAX ADD XML DOCTYPE.

System Variables or Sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

Error Handling

In the event of an error, the command returns an error which can be intercepted using an error-handling method.

SAX ADD XML DOCTYPE

version 2004

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">

See Also

SAX ADD XML COMMENT.

System Variables or Sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

Error Handling

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 (document; data; *)

Parameter	Type		Description
document	DocRef	→	Reference of open document
data	Text | Variable	→	Text or variable to insert in the document
*	*	→	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 ENCODE command. In this case, of course, you must passs a BLOB in data.

By default, the command does not encode special characters (< > ” ’...) contained in the data parameter. To force the encoding of these parameters, simply 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)
```

See Also

SAX GET XML ELEMENT VALUE.

System Variables or 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 (document)

Parameter	Type		Description
document	DocRef	→	Reference of open document

Description

In the XML document referenced by document, the SAX CLOSE XML ELEMENT command writes the statements necessary for closing the last element opened using the SAX OPEN XML ELEMENT command.

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>
```

See Also

SAX OPEN XML ELEMENT, SAX OPEN XML ELEMENT ARRAYS.

SAX GET XML CDATA (document; value)

Parameter	Type		Description
document	DocRef	→	Reference of open document
value	BLOB	←	Element value

Description

The SAX GET XML CDATA command allows you to get 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.

Data is returned as is (it is not modified).

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)
```

See Also

SAX ADD XML CDATA, SAX Get XML node.

System Variables or 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

version 2004

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.

See Also

SAX ADD XML COMMENT, SAX Get XML node.

System Variables or 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 (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.

See Also

SAX Get XML node, SAX SET XML OPTIONS.

System Variables or 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 (document; name; prefix; attrNames; attrValues)

Parameter	Type		Description
document	DocRef	→	Reference of open document
name	String	←	Element name
prefix	String	←	Namespace
attrNames	Array string	←	Attribute names
attrValues	Array string	←	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”

See Also

SAX Get XML node.

System Variables or 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

version 2004

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 text will be returned as is (it will not be modified).

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)
```

See Also

SAX ADD XML ELEMENT VALUE, SAX Get XML node.

System Variables or 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 (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.

Examples

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)
```

See Also

SAX Get XML node.

System Variables or 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 (document) → Longint

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 Start Document	Longint	1
XML Comment	Longint	2
XML Processing Instruction	Longint	3
XML Start Element	Longint	4
XML End Element	Longint	5
XML DATA	Longint	6
XML CDATA	Longint	7
XML Entity	Longint	8
XML End Document	Longint	9

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)
End if
CLOSE DOCUMENT (DocRef)
```

System Variables or 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 (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)
```

See Also

SAX ADD PROCESSING INSTRUCTION, SAX Get XML node.

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 lets you add 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)
```

... will write the following line in the document:

<Book

See Also

SAX CLOSE XML ELEMENT, SAX OPEN XML ELEMENT ARRAYS.

Error Handling

If an invalid character is passed in tag, an error is generated.

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	Array string	→	Array of attribute names
attribValuesArray	Array string	→	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.

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">
```

See Also

SAX CLOSE XML ELEMENT, SAX OPEN XML ELEMENT.

SAX SET XML OPTIONS

version 2004

SAX SET XML OPTIONS (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	→	True (default) = document is indented False = document is not indented

Description

The SAX SET XML OPTIONS 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.

• 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.

• indentation: Indicates whether the document should display indentations corresponding to XML key hierarchies (True) or not (False). By default (if the command is not called or if the parameter is omitted), the document is indented.

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 OPTIONS($DocRef;"UTF-16";True)
```

... will write this line in the document:

```
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
```

See Also

SAX GET XML DOCUMENT VALUES.

SET XSLT 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

Description

The SET XSLT PARAMETER command must be used jointly with the APPLY XSLT TRANSFORMATION command. It allows you to define values of variable parameters placed in an XSL style sheet when the XSLT transformation of an XML document begins. Using this command, it is possible to insert values coming from 4D processes in the XSL style sheets right before using APPLY XSLT TRANSFORMATION.

Notes:
• For more information about XSL transformation (or XSLT), refer to the Overview of XML Commands section.
• The scope of this command is the current process. It must be called in the same process as the associated APPLY XSLT TRANSFORMATION command.

Pass the name of the XSL variable parameter to replace in paramName. This parameter must be present in the XSL style sheet as $toreplace. However, the $ character is not necessary in paramName. For example, if the instruction <xsl:template match=$myvar> is placed in the XSL file, simply pass “myvar” in paramName to set this parameter.

In paramValue, pass the value that you wish to insert instead of the XSL variable in the transformed file. To use the above example, if you pass “title” in paramValue, the XSLT traansformation will take into account the <xsl:template match="title"> instruction (which set “title” elements as subject to style rules).
If the value type is string, you must include it between single quotes (for example 'myvalue') — in addition to the double quotes of the 4D syntax ("'myvalue'").

Note: For a detailed description of XSL language, you can refer to a number of sites on the Internet dedicated to this language. For example, http://xml.org.

To pass several parameters in an XSL style sheet, simply call the SET XSLT PARAMETER command several times. The parameters are “stacked” until the APPLY XSLT TRANSFORMATION call in the same process. Once APPLY XSLT TRANSFORMATION is executed, the “stack” of parameters is automatically deleted.

Example

The following example defines two XSL parameters then transforms the document mydoc.xml into an HTML file using the style sheet mysheet.xsl:

```
SET XSLT PARAMETER("varstyle";"'bold'")
SET XSLT PARAMETER("varcolor";"'blue'")
$xmldoc:="mydoc.xml"
$xslsheet:="mysheet.xsl"
$htmldoc:="mydoc.html"
APPLY XSLT TRANSFORMATION($xmldoc;$xslsheet;$htmldoc)
```

See Also

APPLY XSLT TRANSFORMATION, GET XSLT ERROR.

System Variables or Sets

If the command has been executed correctly, the system variable OK is set to 1. Otherwise, it is set to 0.

67

Error Codes

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.

See Also

ON ERR CALL.

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
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
1006	Program interrupted by user—user pressed Alt-click (Windows) or Option-click (Mac OS)
-1	Unknown entry point requested by a Plug-In
-9750	The source form is not editable.
-9751	The source form is not accessible by the user.
-9752	The user form cannot be created.
-9753	The source form does not exist.
-9754	This command cannot be used from a dialog window.
-9755	The user form does not have a name.
-9756	There is no user structure file.
-9757	The user form does not exist.
-9758	The user form already exists.
-9759	The Object Library could not be opened.
-9800	One of the processes modified the access rights.
-9850	Invalid area parameter passed to an external command.
-9851	Invalid parameter number 1.
-9852	Invalid parameter number 2.
-9853	Invalid parameter number 3.
-9854	Invalid parameter number 4.
-9855	Invalid parameter number 5.
-9910	Soap fault.
-9911	Parser fault.
-9912	HTTP fault.
-9913	Network fault.

-9914 Internal fault.
-9915 The document's reference is invalid.
-9916 The element is not open.
-9917 The type of the array passed in parameter is invalid.
-9918 The name of the element is invalid.
-9919 This encoding is not supported.
-9920 The type of the node is invalid
-9925 The referenced element is null.
-9926 The referenced element is invalid.
-9927 The referenced element is not the “root”.
-9928 The name of the element is unknown.
-9929 The index for this element is invalid.
-9930 There is no attribute with this name for this element.
-9931 The index for this attribute is invalid.
-9932 The XML DLL is not loaded.
-9933 The XML file is not valid.
-9934 The XML file is not well-formed.
-9935 The XML file is not valid or is not well-formed.
-9937 Password System is locked by another user.
-9938 The current record has been changed from within the trigger.
-9939 External routine not found.
-9940 4D Extension initialization failed.
-9941 Unknown EX_GESTALT selector.
-9942 4D Client licensing scheme is incompatible with this version of 4D Server.
-9943 4D Connectivity Plug-ins version error.
-9944 The user does not belong to the 4D Open access group.
-9945 CD-ROM 4D Runtime error; writing operations are not allowed.
-9946 Unable to clear the named selection because it does not exist.
-9947 The “Allow 4D Open connections” check box has not been selected.
-9948 A modal dialog is activated.
-9949 License or privilege error.
-9950 Invalid data segment number.
-9951 This field has no relation.
-9952 Invalid data segment header.
-9953 There is no Log file.
-9954 There is no current record.
-9955 QuickTime is not installed.
-9956 Versions of 4D Client and 4D Server are different.
-9957 The choice list is locked.
-9958 The process could not be started.
-9959 The backup process has already been started by another user or process.
-9960 4D Backup is not installed on the server.

-9961 The backup process is not currently running.
-9962 The backup cannot be run because the server is shutting down.
-9963 Invalid record number requested by a workstation.
-9964 Bad sort definition table sent by a workstation.
-9965 Bad search definition table sent by a workstation.
-9966 Invalid type requested by a workstation.
-9967 The record could not be modified because it could not be loaded.
-9968 Invalid selected record number requested by workstation.
-9969 Invalid field type requested by workstation.
-9970 Field is not indexed.
-9971 Field number is out of range requested by workstation.
-9972 Table number is out of range requested by workstation.
-9973 The TRIC resources are not the same.
-9974 Record has already been deleted.
-9975 Transaction index page could not be loaded.
-9976 Backup in progress; no modification allowed.
-9977 The selection does not exist.
-9978 Bad user password.
-9979 Unknown user.
-9980 The file cannot be created because the structure is locked.
-9981 Invalid field name/field number definition table sent by the workstation.
-9982 The record was not loaded because it is not in the selection on the workstation.
-9983 The same external package is installed twice.
-9984 Transaction has been cancelled because of a duplicated index key error.
-9985 Recursive integrity.
-9986 Record locked during an automatic deletion action.
-9987 Some other records are already related to this record.
-9988 The form cannot be loaded. Either the form or the structure is damaged.
-9989 Invalid structure (database needs to be repaired).
-9990 Time-out error.
-9991 Privileges error.
-9992 Wrong password.
-9993 Menu bar is damaged (database needs to be repaired).
-9994 Serial communication interrupted by the user—user pressed Ctrl-Alt-Shift (Windows) or Command-Option-Shift (Mac OS).
-9995 Demo limit has been reached.
-9996 Stack is full (too much recursion or nested calls).
-9997 Maximum number of records has been reached.
-9998 Duplicated key.
-9999 No more room to save the record. (see note 4)
-10500 Invalid record address (database needs to be repaired).
-10501 Invalid index page (index needs to be repaired).

-10502 Invalid record structure (data file needs to be repaired).
-10503 Record # is out of range (during GOTO RECORD, or data file needs to be repaired). (see note 3)
-10504 Index page # is out of range (index needs to be repaired).
-10505 Client and server have version numbers that are incompatible.
-10506 Limit of the Standard Edition version.
-10507 This version does not allow a compiled database to be opened.
-10508 Project method not found.
-10509 Can't open database "".
-10510 Can't load component "".
-10511 Can't call command "" from a component.
-10600 This BLOB could not be read. It may be damaged.

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.

See Also

ON ERR CALL.

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. 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 You must allocate more memory to 4D.
28	Stack has moved into the application heap. You must allocate more memory to 4D.

The following table describes the errors that can occur with a network connection.

Code	Description
-10001	The actual connection to the database has been disrupted.
-10002	The connection for this process has been disrupted.
-10003	Bad connection parameters.
-10020	No server was selected while using OP Select 4D server.
-10021	No server was found while using OP Find 4D server.
-10030	Desynchronization has occured during the write cycle.
-10031	Desynchronization has occured during the read cycle.
-10033	Incorrect data size during read cycle.
-10050	Unknown option in Get/SetOption.
-10051	Incorrect value in Get/SetOption.

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. In this list, the word “file” indicates a document on disk and not a file in your database structure.

Code	Description
-33	File directory full. You cannot create new files on disk.
-34	Disk is full. There is no more room available on the disk.
-35	Specified volume doesn’t exist.
-36	I/O error. There is probably a bad block on the disk.
-37	Bad filename or volume name.
-38	Tried to read or write to a file that is not open.
-39	Logical end-of-file reached during read operation.
-40	Attempt to position before start of file.
-41	Not enough memory to open a new file on the disk.
-42	Too many files open at the same time.
-43	File not found.
-44	Volume is locked by a hardware setting.
-45	File is locked.
-46	Volume is locked by an application.
-47	Tried to access a file that has been deleted.
-48	Tried to rename a file with the name of an already deleted file.
-49	Tried to open a file already open with write permission.
-51	Tried to access a document with an invalid document reference number.
-52	Internal file manager error (position of file marker is lost).
-53	Volume not on line.
-54	Attempt to open locked file for writing.
-57	Tried to work with a non-Macintosh disk.
-58	Error in the external file system.
-60	Bad master directory block. Your disk is damaged.
-61	Read/write permission doesn’t allow writing.
-64	There is a hardware problem with the disk (bad installation, incorrect formatting,...).
-84	There is a hardware problem with the disk (bad installation, incorrect formatting,...).
-120	Tried to access a file by using a pathname that specifies a non existing directory.
-121	An access path could not be created.
-124	Tried to access a disconnected shared volume.

See Also

ON ERR CALL.

The following table lists the error codes returned by the Operating System Memory Manager.

Code	Description
-108	Not enough memory to perform an operation. Give more memory to your 4D application.
-109	Internal Memory problem. Memory is probably logically corrupted. Exit as soon as possible. Restart your machine and reopen the database.
-111	Internal Memory problem. Memory is probably logically corrupted. Exit as soon as possible. Restart your machine and reopen the database. (*)
-117	Internal Memory problem. Memory is probably logically corrupted. Exit as soon as possible. Restart your machine and reopen the database.

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.

See Also

ON ERR CALL.

The following table lists the error codes returned by the Operating System Printing Manager. These codes can be returned during printing.

Code	Description
-1	Problem saving file to be printed
-27	Problem opening or closing connection with printer
-128	Printing interrupted by the user
-193	Resource file not found
-4100	Printer connection has been interrupted
-4101	Printer busy or not connected
-8150	A LaserWriter is not selected
-8151	The printer has been initialized with a different driver version
-8192	LaserWriter time-out

See Also

ON ERR CALL.

The following table lists the error codes returned by the Operating System Resource Manager.

Code	Description
-1	Resource file could not be opened
-192	Resource not found
-193	Resource map is damaged (file needs to be repaired)
-194	Resource could not be added
-196	Resource could not be deleted

See Also

ON ERR CALL.

OS Serial Ports Manager Errors

The following table lists error codes returned by the Operating System Serial Ports Manager.

Code	Description
-28	There is no open serial port

See Also

ON ERR CALL.

OS Sound Manager Errors

The following table lists the codes returned by the Operating System Sound Manager.

Code	Description
-203	Too many sound commands
-204	The sound resource cannot be loaded
-205	The sound channel is logically corrupted
-206	The format of the sound resource is wrong
-207	Not enough memory to perform the sound
-209	The sound channel is busy

See Also

ON ERR CALL.

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

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 SQL ERROR command.

Generic errors

1001	INVALID ARGUMENT
1002	INVALID INTERNAL STATE
1003	NOT RUNNING

Semantic errors

1101	TABLE DOES NOT EXIST
1102	COLUMN DOES NOT EXIST
1103	TABLE NOT DECLARED IN FROM CLAUSE
1104	AMBIGUOUS COLUMN NAME
1105	TABLE ALIAS SAME AS TABLE NAME
1106	DUPLICATE TABLE ALIAS
1107	DUPLICATE TABLE IN FROM CLAUSE
1108	INCOMPATIBLE TYPES
1109	INVALID ORDER BY INDEX
1110	WRONG AMOUNT OF PARAMETERS
1111	INCOMPATIBLE PARAMETER TYPE
1112	UNKNOWN FUNCTION
1113	DIVISION BY ZERO
1114	ORDER BY INDEX NOT ALLOWED
1115	DISTINCT NOT ALLOWED
1116	NESTED COLUMN FUNCTIONS NOT ALLOWED
1117	COLUMN FUNCTIONS NOT ALLOWED
1118	CAN NOT MIX COLUMN AND SCALAR OPERATIONS
1119	INVALID GROUP BY INDEX
1120	GROUP BY INDEX NOT ALLOWED
1121	GROUP BY NOT ALLOWED WITH SELECT ALL
1122	NOT A COLUMN EXPRESSION
1123	NOT A GROUPING COLUMN IN AGGREGATE ORDER BY
1124	MIXED LITERAL TYPES IN PREDICATE
1125	LIKE ESCAPE IS NOT ONE CHAR
1126	BAD LIKE ESCAPE CHAR
1127	UNKNOWN ESCAPE SEQUENCE IN LIKE
1128	COLUMNS FROM MORE THAN ONE QUERY IN COLUMN FUNCTION
1129	SCALAR EXPRESSION WITH GROUP BY
1130	SUBQUERY HAS MORE THAN ONE COLUMN
1131	SUBQUERY MUST HAVE ONE ROW

1132	INSERT VALUE COUNT DOES NOT MATCH COLUMN COUNT
1133	DUPLICATE COLUMN IN INSERT
1134	COLUMN DOES NOT ALLOW NULLS
1135	DUPLICATE COLUMN IN UPDATE
1136	TABLE ALREADY EXISTS
1137	DUPLICATE COLUMN IN CREATE TABLE
1138	DUPLICATE COLUMN IN COLUMN LIST
1139	MORE THAN ONE PRIMARY KEY NOT ALLOWED
1140	AMBIGUOUS FOREIGN KEY NAME
1141	COLUMN COUNT MISMATCH IN FOREIGN KEY
1142	COLUMN TYPE MISMATCH IN FOREIGN KEY
1143	FAILED TO FIND MATCHING PRIMARY COLUMN
1144	UPDATE AND DELETE CONSTRAINTS MUST BE THE SAME
1145	FOREIGN KEY DOES NOT EXIST
1146	INVALID LIMIT VALUE IN SELECT
1147	INVALID OFFSET VALUE IN SELECT
1148	PRIMARY KEY DOES NOT EXIST
1149	FAILED TO CREATE FOREIGN KEY
1150	FIELD IS NOT IN PRIMARY KEY
1151	FIELD IS NOT UPDATEABLE

Implementation errors

1203	FUNCTIONALITY IS NOT IMPLEMENTED
1204	FAILED TO CREATE NEW RECORD
1205	FAILED TO UPDATE FIELD
1206	FAILED TO DELETE RECORD
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

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 PARAMETER VALUE NOT SPECIFIED
1416 COLUMN See Also PARAMETERS FROM DIFFERENT TABLES
1417 EMPTY STATEMENT
1418 FAILED TO UPDATE VARIABLE
1419 FAILED TO GET TABLE See Also
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

Date parsing 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

Date formatting error

1551 FAILED

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
1607	NO VALID TOKENS

Cacheable errors

2000	CACHEABLE NOT INITIALIZED
2001	VALUE ALREADY CACHED
2002	CACHED VALUE NOT FOUND

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

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.

See Also

ON ERR CALL.

68

ASCII Codes

ASCII Code Tables

• The standard ASCII codes, 0 through 127, are common to Windows and Macintosh. These standard ASCII codes are listed in ASCII Codes 0..63 and ASCII Codes 64..127.

• The ASCII codes 128 through 255 are different on Windows and Macintosh. In order to maintain platform independence, the Windows version of 4D automatically converts ASCII codes from Windows to Macintosh ASCII maps when characters are entering the 4D environment (Data entry, Edit/Paste, Import, etc.) and from Macintosh to Windows ASCII maps when characters are leaving the 4D environment (Edit/Cut or Copy, Export, etc.).

The ASCII codes 128 through 255 are listed in ASCII Codes 128..191 and ASCII Codes 192..255.

Understanding ASCII Codes and 4D

When the database is operating in ASCII compatibility mode, on both Macintosh and Windows, the internal database engine and the 4D language work with the Macintosh extended ASCII set (for more information about ASCII compatibility mode and Unicode mode, please refer to the About Unicode section). When you enter data using the keyboard (adding records, editing procedures, etc.), 4D uses the internal Altura ASCII conversion scheme to convert what comes from the keyboard (expressed using the Windows set) to the Macintosh set. For example, to enter an “é”, you type ALT+0233, and 4D stores ASCII code 142 in the record. This is transparent to the end user, because when you create a search, you actually type (in the Search editor) the value for which you are looking. Therefore, the value that you typed (ALT+0233) is also translated into ASCII code 142, and you find the value.

The codes work the same when you type ALT+0233 in the Procedure editor. However, to look for a character using its ASCII code, you use the Macintosh ASCII code of the character.

For example:

```
QUERY (...; [MyFile]MyField="é") ` é is Alt+0233
```

is the same as:

```
QUERY (...;[MyFile]MyField=Char(142)) ` é is ASCII 142
```

See Also

Character code, ISO to Mac, Mac to ISO, Mac to Win, ON EVENT CALL, Win to Mac.

The standard ASCII codes (0 through 127) are common to Windows and Macintosh.

ASCII Codes 0..63 ASCII Codes 64..127 ASCII Codes 128..191 ASCII Codes 192..255

ASCII		Macintosh or Windows
Dec	Hex	Result
0	0	*NUL*
1	1	*SOH*
2	2	*STX*
3	3	*ETX*
4	4	*EOT*
5	5	*ENQ*
6	6	*ACK*
7	7	*BEL*
8	8	*BS*
9	9	*HT*
10	A	*LF*
11	B	*VT*
12	C	*FF*
13	D	*CR*
14	E	*SO*
15	F	*SI*

ASCII		Macintosh or Windows
Dec	Hex	Result
16	10	*DLE*
17	11	*DC1*
18	12	*DC2*
19	13	*DC3*
20	14	*DC4*
21	15	*NAK*
22	16	*SYN*
23	17	*ETB*
24	18	*CAN*
25	19	*EM*
26	1A	*SUB*
27	1B	*ESC*
28	1C	*FS*
29	1D	*GS*
30	1E	*RS*
31	1F	*US*

ASCII		Macintosh or Windows
Dec	Hex	Result
32	20	sp
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/

ASCII		Macintosh or Windows
Dec	Hex	Result
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

ASCII Codes 0..63 ASCII Codes 64..127 ASCII Codes 128..191 ASCII Codes 192..255

See Also

Char, ISO to Mac, Mac to ISO, Mac to Win, ON EVENT CALL, Win to Mac.

The standard ASCII codes (0 through 127) are common to Windows and Macintosh.

ASCII Codes 0..63 ASCII Codes 64..127 ASCII Codes 128..191 ASCII Codes 192..255

ASCII		Macintosh or Windows	ASCII		Macintosh or Windows
Dec	Hex	Result	Dec	Hex	Result
64	40	@	80	50	P
65	41	A	81	51	Q
66	42	B	82	52	R
67	43	C	83	53	S
68	44	D	84	54	T
69	45	E	85	55	U
70	46	F	86	56	V
71	47	G	87	57	W
72	48	H	88	58	X
73	49	I	89	59	Y
74	4A	J	90	5A	Z
75	4B	K	91	5B	[
76	4C	L	92	5C	\
77	4D	M	93	5D	]
78	4E	N	94	5E	^
79	4F	O	95	5F	_

ASCII		Macintosh or Windows
Dec	Hex	Result
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

ASCII		Macintosh or Windows
Dec	Hex	Result
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	DEL

ASCII Codes 0..63 ASCII Codes 64..127 ASCII Codes 128..191 ASCII Codes 192..255

See Also

Ascii, ISO to Mac, Mac to ISO, Mac to Win, ON EVENT CALL, Win to Mac.

The following tables list the characters displayed by 4D for each ASCII code, on Macintosh and Windows. In addition, the tables present the key combination required to produce each character, using a US keyboard.

ASCII Codes 0..63 ASCII Codes 64..127 ASCII Codes 128..191 ASCII Codes 192..255

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
128	80	Ä	Option-u, Shift-a	Ä	Alt+0196
129	81	Å	Option-Shift-a	Å	Alt+0197
130	82	Ç	Option-Shift-c	Ç	Alt+0199
131	83	É	Option-e, Shift-e	É	Alt+0201
132	84	Ñ	Option-n, Shift-n	Ñ	Alt+0209
133	85	Ö	Option-u, Shift-o	Ö	Alt+0214
134	86	Ü	Option-u, Shift-u	Ü	Alt+0220
135	87	á	Option-e, a	á	Alt+0225
136	88	à	Option-`, a	à	Alt+0224
137	89	â	Option-i, a	â	Alt+0226
138	8A	ä	Option-u, a	ä	Alt+0228
139	8B	ã	Option-n, a	ã	Alt+0227
140	8C	å	Option-a	å	Alt+0229
141	8D	ç	Option-c	ç	Alt+0231
142	8E	é	Option-e, e	é	Alt+0233
143	8F	è	Option-`, e	è	Alt+0232

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
144	90	ê	Option-i, e	ê	Alt+0234
145	91	ë	Option-u, e	ë	Alt+0235
146	92	í	Option-e, i	Í	Alt+0237
147	93	ì	Option-`, i	Ì	Alt+0236
148	94	î	Option-i, i	Î	Alt+0238
149	95	ï	Option-u, i	Ï	Alt+0239
150	96	ñ	Option-n, n	ñ	Alt+0241
151	97	ó	Option-e, o	ó	Alt+0243
152	98	ò	Option-`, o	Ò	Alt+0242
153	99	ô	Option-i, o	Ô	Alt+0244
154	9A	ö	Option-u, o	Ö	Alt+0246
155	9B	õ	Option-n, o	õ	Alt+0245
156	9C	ú	Option-e, u	ú	Alt+0250
157	9D	ù	Option-`, u	ù	Alt+0249
158	9E	û	Option-i, u	û	Alt+0251
159	9F	ü	Option-u, u	ü	Alt+0252

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
160	A0	†	Shift-Option-7		
161	A1	°	Shift-Option-7	°	Alt+0176
162	A2	¢	Option-4	¢	Alt+0162
163	A3	£	Option-3	£	Alt+0163
164	A4	§	Option-6	§	Alt+0167
165	A5	•	Option-8		
166	A6	¶	Option-7	¶	Alt+0182
167	A7	ß	Option-s	ß	Alt+0223
168	A8	®	Option-r	®	Alt+0174
169	A9	©	Option-g	©	Alt+0169
170	AA	™	Option-2	™	Alt-0153
171	AB	´	Shift-Option-e	´	Alt+0145
172	AC	¨	Shift-Option-u	¨	Alt+0168
173	AD	≠	Option-=		
174	AE	Æ	Shift-Option-"	Æ	Alt+0198
175	AF	Ø	Shift-Option-o	Ø	Alt+0216

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
176	B0	∞	Option-5		
177	B1	±	Shift-Option-=	±	Alt+0177
178	B2	≤	Option-,		
179	B3	≥	Option-. (period)		
180	B4	¥	Option-y	¥	Alt+0165
181	B5	µ	Option-m	µ	Alt+0181
182	B6	∂	Option-d		
183	B7	∑	Option-w		
184	B8	∏	Shift-Option-p		
185	B9	π	Option-p		
186	BA	∫	Option-b		
187	BB	ª	Option-9	ª	Alt+0170
188	BC	º	Option-0 (zero)	º	Alt+0186
189	BD	Ω	Option-z		
190	BE	æ	Option-'	æ	Alt+0230
191	BF	ø	Option-o	ø	Alt+0248

ASCII Codes 0..63 ASCII Codes 64..127 ASCII Codes 128..191 ASCII Codes 192..255

Note: The cells in the Windows column that are greyed out denote characters that are not available on Windows or that are different from the Macintosh characters.

See Also

Char, ISO to Mac, Mac to ISO, Mac to Win, ON EVENT CALL, Win to Mac.

The following tables list the characters displayed by 4D for each ASCII code, on Macintosh and Windows. In addition, the tables present the key combination required to produce each character, using a US keyboard.

ASCII Codes 0..63 ASCII Codes 64..127 ASCII Codes 128..191 ASCII Codes 192..255

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
192	C0	¿	Shift-Option-?	¿	Alt+0191
193	C1	¡	Option-1	¡	Alt+0161
194	C2	¬	Option-l (L)	¬	Alt+0172
195	C3	√	Option-v		
196	C4	ƒ	Option-f		
197	C5	≈	Option-x		
198	C6	Δ	Option-j		
199	C7	«	Option-\	«	Alt+0171
200	C8	»	Shift-Option-\	»	Alt+0187
201	C9	…	Option-;	…	Alt+0133
202	CA	(space)	Spacebar	(space)	Alt+0160
203	CB	À	Option-`, Shift-a	À	Alt+0192
204	CC	Ã	Option-n, Shift-a	Ã	Alt+0195
205	CD	Õ	Option-n, Shift-o	Õ	Alt+0213
206	CE	Œ	Shift-Option-q		
207	CF	œ	Option-q		

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
208	D0	–	Option--(dash)		
209	D1	—	Shift-Option-(dash)		
210	D2	“	Option-[	“	Alt+0147
211	D3	”	Shift-Option-[	”	Alt+0148
212	D4	‘	Option-]	‘	Alt+0145
213	D5	’	Shift-Option-]	’	Alt+0146
214	D6	÷	Option-/	÷	Alt+0247
215	D7	◊	Shift-Option-v		
216	D8	ÿ	Option-u, y	ÿ	Alt+0255
217	D9	Ÿ	Option-u, Shift-y		
218	DA	⁄	Shift-Option-1		
219	DB	¤	Shift-Option-2	¤	Alt+0164
220	DC	‹	Shift-Option-3		
221	DD	›	Shift-Option-4		
222	DE	fi	Shift-Option-5		
223	DF	fl	Shift-Option-6		

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
224	E0	‡	Shift-Option-7		
225	E1	·	Shift-Option-9	·	Alt+0183
226	E2	‚	Shift-Option-0		
227	E3	„	Shift-Option-w		
228	E4	‰	Shift-Option-r		
229	E5	Â	Option-i, Shift-a	Â	Alt+0194
230	E6	Ê	Option-i, Shift-e	Ê	Alt+0202
231	E7	Á	Option-e, Shift-a	Á	Alt+0193
232	E8	Ë	Option-u, Shift-e	Ë	Alt+0203
233	E9	È	Option-`, Shift-e	È	Alt+0200
234	EA	í	Shift-Option-s	Í	Alt+0205
235	EB	î	Shift-Option-d	Î	Alt+0206
236	EC	ï	Shift-Option-f	Ï	Alt+0207
237	ED	ì	Option-`, Shift-i	Ì	Alt+0204
238	EE	ó	Shift-Option-h	Ó	Alt+0211
239	EF	ô	Shift-Option-j	Ô	Alt+0212

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
240	F0		Shift-Option-k		
241	F1	Ò	Shift-Option-l (L)	Ò	Alt+0210
242	F2	Ú	Shift-Option-;	Ú	Alt+0218
243	F3	Û	Option-i, Shift-u	Û	Alt+0219
244	F4	Ù	Option-`, Shift-u	Ù	Alt+0217
245	F5	ı	Shift-Option-b		
246	F6	ˆ	Shift-Option-i		
247	F7	˜	Shift-Option-n		
248	F8	¯	Shift-Option-,	¯	Alt+0175
249	F9	˘	Shift-Option-.		
250	FA	˙	Option-h		
251	FB	°	Option-k		
252	FC	¸	Shift-Option-z	¸	Alt+0184
253	FD	˝	Shift-Option-g		
254	FE	˛	Shift-Option-x		
255	FF	ˇ	Shift-Option-t		

ASCII Codes 0..63 ASCII Codes 64..127 ASCII Codes 128..191 ASCII Codes 192..255

Note: The cells in the Windows column that are greyed out denote characters that are not available on Windows or that are different from the Macintosh characters.

See Also

Ascii, ISO to Mac, Mac to ISO, Mac to Win, ON EVENT CALL, Win to Mac.

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.

Code	Key
3	Enter
13	Return
8	Backspace or Delete
9	Tab
27	Escape or Clear
127	Del
5	Help
1	Home
4	End
11	Page Up
12	Page Down
28	Left Arrow
29	Right Arrow
30	Up Arrow
31	Down Arrow

See Also

ON EVENT CALL.

69

Command Syntax

The first column indicates the number for each command, used more particularly by the Command name command.

A

156 ABORT
99 Abs (number) → Number
269 ACCEPT
303 ACCUMULATE (data{; data2; ...; dataN})
346 Activated → Boolean
361 ADD DATA SEGMENT
56 ADD RECORD ({aTable}{; }{*})
202 ADD SUBRECORD (subtable; form{; *})
393 Add to date (date; years; months; days) → Date
119 ADD TO SET ({table; }set)
31 After → Boolean
41 ALERT (message{; ok button title})
47 ALL RECORDS {(table)}
109 ALL SUBRECORDS (subtable)
403 APPEND DATA TO PASTEBOARD (dataType; data)
265 Append document (document{; fileType}) → DocRef
411 APPEND MENU ITEM (menu; itemText{; subMenu{; process}})
911 APPEND TO ARRAY (array; value)
376 APPEND TO LIST (list; itemText; itemRef{; sublist; expanded})
491 Application file → String
494 Application type → Long Integer
493 Application version {(*)} → String
70 APPLY TO SELECTION (aTable; statement)
73 APPLY TO SUBSELECTION (subtable; statement)
882 APPLY XSLT TRANSFORMATION (xmlSource; xslSheet; result{; compileSheet})
20 Arctan (number) → Number
223 ARRAY BOOLEAN (arrayName; size{; size2})
224 ARRAY DATE (arrayName; size{; size2})
220 ARRAY INTEGER (arrayName; size{; size2})

221 ARRAY LONGINT (arrayName; size{; size2})
279 ARRAY PICTURE (arrayName; size{; size2})
280 ARRAY POINTER (arrayName; size{; size2})
219 ARRAY REAL (arrayName; size{; size2})
218 ARRAY STRING (strLen; arrayName; size{; size2})
222 ARRAY TEXT (arrayName; size{; size2})
287 ARRAY TO LIST (array; list{; itemRefs})
261 ARRAY TO SELECTION (array; field{; array2; field2; ...; arrayN; fieldN})
512 ARRAY TO STRING LIST (strings; resID{; resFile})
786 AUTHENTICATE WEB SERVICE (name; password)
2 Average (series) → Number

B

887 BACKUP
151 BEEP
29 Before → Boolean
198 Before selection {(table)} → Boolean
199 Before subselection (subtable) → Boolean
948 Begin SQL
717 BEST OBJECT SIZE ({*; }object; bestWidth; bestHeight{; maxWidth})
536 BLOB PROPERTIES (blob; compressed{; expandedSize{; currentSize}})
605 BLOB size (blob) → Longint
526 BLOB TO DOCUMENT (document; blob{; *})
549 BLOB to integer (blob; byteOrder{; offset}) → Number
557 BLOB to list (blob{; offset}) → ListRef
551 BLOB to longint (blob; byteOrder{; offset}) → Number
682 BLOB TO PICTURE (pictureBlob; picture)
553 BLOB to real (blob; realFormat{; offset}) → Real
555 BLOB to text (blob; textFormat{; offset{; textLength}}) → Text
850 BLOB TO USERS (users)
533 BLOB TO VARIABLE (blob; variable{; offset})
646 BOOLEAN ARRAY FROM SET (booleanArr{; set})
302 BREAK LEVEL (level{; pageBreak})
326 BRING TO FRONT (process)
871 BUILD APPLICATION {(projectName)}
194 BUTTON TEXT ({*; }object; buttonText)

C

541 COPY DOCUMENT (sourceName; destinationName{; *})
626 Copy list (list) → ListRef
331 COPY NAMED SELECTION ({table; }name)
600 COPY SET (srcSet; dstSet)
18 Cos (number) → Number
907 Count in array (array; value) → Longint
380 Count list items ({*; }list{; *}) → Longint
405 Count menu items (menu{; process}) → Number
404 Count menus {(process)} → Number
259 Count parameters → Number
437 Count screens → Longint
335 Count tasks → Integer
343 Count user processes → Integer
342 Count users → Integer
694 CREATE ALIAS (targetPath; aliasPath)
313 CREATE DATA FILE (accessPath)
266 Create document (document{; fileType}) → DocRef
140 CREATE EMPTY SET ({table; }set)
475 CREATE FOLDER (folderPath)
966 CREATE INDEX (theTable; fieldsArray; indexType; indexName{; *})
408 Create menu {(menu)} → MenuRef
68 CREATE RECORD {(table)}
65 CREATE RELATED ONE (field)
496 Create resource file (resFilename{; fileType{; *}}) → DocRef
640 CREATE SELECTION FROM ARRAY (table; recordArray{; selectionName})
116 CREATE SET ({table; }set)
641 CREATE SET FROM ARRAY (table; recordsArray{; setName})
72 CREATE SUBRECORD (subtable)
679 CREATE THUMBNAIL (source; dest{; width{; height{; mode{; depth}}}})
808 CREATE USER FORM (aTable; form; userForm)
33 Current date {(*)} → Date
363 Current default table → Pointer
276 Current form page → Number
627 Current form table → Pointer
827 Current form window → WinRef
483 Current machine → String
484 Current machine owner → String
684 Current method name → String
322 Current process → Number
178 Current time {(*)} → Time

182 Current user → String
334 CUT NAMED SELECTION ({table; }name)
604 C_BLOB ({method; }variable{; variable2; ...; variableN})
305 C_BOOLEAN ({method; }variable{; variable2; ...; variableN})
307 C_DATE ({method; }variable{; variable2; ...; variableN})
352 C_GRAPH ({method; }variable{; variable2; ...; variableN})
282 C_INTEGER ({method; }variable{; variable2; ...; variableN})
283 C_LONGINT ({method; }variable{; variable2; ...; variableN})
286 C_PICTURE ({method; }variable{; variable2; ...; variableN})
301 C_POINTER ({method; }variable{; variable2; ...; variableN})
285 C_REAL ({method; }variable{; variable2; ...; variableN})
293 C_STRING ({method; }size; variable{; variable2; ...; variableN})
284 C_TEXT ({method; }variable{; variable2; ...; variableN})
306 C_TIME ({method; }variable{; variable2; ...; variableN})

D

490 Data file {(segment)} → String
527 DATA SEGMENT LIST (Segments)
369 Database event → Longint
102 Date (dateString) → Date
114 Day number (aDate) → Number
23 Day of (date) → Number
347 Deactivated → Boolean
9 Dec (number) → Number
896 DECODE (blob)
690 DECRYPT BLOB (toDecrypt; sendPubKey{; recipPrivKey})
46 DEFAULT TABLE (aTable)
323 DELAY PROCESS (process; duration)
159 DELETE DOCUMENT (document)
693 DELETE FOLDER (folder)
228 DELETE FROM ARRAY (array; where{; howMany})
560 DELETE FROM BLOB (blob; offset; len)
624 DELETE FROM LIST ({*; }list; itemRef | *{; *})
967 DELETE INDEX (fieldPtr | indexName{; *})
830 DELETE LISTBOX COLUMN ({*; }object; colPosition{; number})
914 DELETE LISTBOX ROW ({*; }object; position)
413 DELETE MENU ITEM (menu; menuItem{; process})
58 DELETE RECORD {(table)}
501 DELETE RESOURCE (resType; resID{; resFile})

66 DELETE SELECTION {(table)}
232 Delete string (source; where; numChars) → String
96 DELETE SUBRECORD (subtable)
615 DELETE USER (userID)
810 DELETE USER FORM (aTable; form; userForm)
1044 DESCRIBE QUERY EXECUTION (status)
40 DIALOG ({aTable; }form; *)
122 DIFFERENCE (set; subtractSet; resultSet)
193 DISABLE BUTTON ({*; }object)
150 DISABLE MENU ITEM (menu; menuItem{; process})
910 DISPLAY NOTIFICATION (title; text{; duration})
105 DISPLAY RECORD {(table)}
59 DISPLAY SELECTION ({aTable}{; selectMode{; enterList{; *{; *}}}})
897 Displayed line number → Longint
339 DISTINCT VALUES (aField; array)
529 Document creator (document) → String
474 DOCUMENT LIST (pathname; documents)
525 DOCUMENT TO BLOB (document; blob{; *})
528 Document type (document) → String
722 DOM CLOSE XML (elementRef)
727 DOM Count XML attributes (elementRef) → Longint
726 DOM Count XML elements (elementRef; elementName) → Longint
865 DOM Create XML element (elementRef; xPath{; attrName{; attrValue}}{; attrName2; attrValue2; ...; attrNameN; attrValueN}) → String
861 DOM Create XML Ref (root{; nameSpace{; nameSpaceName{; nameSpaceValue}}} {; nameSpaceName2; nameSpaceValue2; ...; nameSpaceNameN; nameSpaceValueN}) → String
862 DOM EXPORT TO FILE (elementRef; filePath)
1017 DOM EXPORT TO PICTURE (elementRef; pictVar{; exportType})
863 DOM EXPORT TO VAR (elementRef; vXmlVar)
864 DOM Find XML element (elementRef; xPath{; arrElementRefs}) → elementRef
1010 DOM Find XML element by ID (elementRef; id) → String
723 DOM Get first child XML element (elementRef{; childElemName{; childElemValue}}) → String
925 DOM Get last child XML element (elementRef{; childElemName{; childElemValue}}) → String
724 DOM Get next sibling XML element (elementRef{; siblingElemName{; siblingElemValue}}) → String

923 DOM Get parent XML element (elementRef{; parentElemName{; parentElemValue}}) → String
924 DOM Get previous sibling XML element (elementRef{; siblingElemName {; siblingElemValue}}) → String
729 DOM GET XML ATTRIBUTE BY INDEX (elementRef; attribIndex; attribName; attribValue)
728 DOM GET XML ATTRIBUTE BY NAME (elementRef; attribName; attribValue)
725 DOM Get XML element (elementRef; elementName; index; elementValue) → String
730 DOM GET XML ELEMENT NAME (elementRef; elementName)
731 DOM GET XML ELEMENT VALUE (elementRef; elementValue{; cDATA})
721 DOM Get XML information (elementRef; xmlInfo) → String
719 DOM Parse XML source (document{; validation{; dtd | schema}}) → String
720 DOM Parse XML variable (variable{; validation{; dtd}}) → String
869 DOM REMOVE XML ELEMENT (elementRef)
866 DOM SET XML ATTRIBUTE (elementRef; attrName; attrValue{; attrName2; attrValue2; ...; attrNameN; attrValueN})
867 DOM SET XML ELEMENT NAME (elementRef; elementName)
868 DOM SET XML ELEMENT VALUE (elementRef{; xPath; elementValue{; *})
859 DOM SET XML OPTIONS (elementRef; encoding{; standalone{; indentation}})
607 DRAG AND DROP PROPERTIES (srcObject; srcElement; srcProcess)
452 DRAG WINDOW
608 Drop position {(columnNumber)} → Number
225 DUPLICATE RECORD {(table)}
30 During → Boolean
1006 Dynamic pop up menu (menu{; default{; xCoord{; yCoord}}}) → ItemRef

E

281 EDIT ACCESS
807 EDIT FORM (aTable; form{; userForm{; library}})
806 EDIT FORMULA (table; formula)
870 EDIT ITEM ({*; }object{; item})
192 ENABLE BUTTON ({*; }object)
149 ENABLE MENU ITEM (menu; menuItem{; process})
895 ENCODE (blob)
689 ENCRYPT BLOB (toEncrypt; sendPrivKey{; recipPubKey})
36 End selection {(table)} → Boolean
949 End SQL
37 End subselection (subtable) → Boolean
160 ERASE WINDOW {(window)}
676 Euro converter (value; fromCurrency; toCurrency) → Real

63 EXECUTE FORMULA (statement)
1007 EXECUTE METHOD (methodName; result | *{; param}{; param2; ...; paramN})
651 EXECUTE ON CLIENT (clientName; methodName{; param}{; param2; ...; paramN})
373 Execute on server (procedure; stack{; name{; param{; param2; ...; paramN}{; *}}})
→ Number
21 Exp (number) → Number
535 EXPAND BLOB (blob)
666 EXPORT DATA (fileName{; project{; *}})
84 EXPORT DIF ({aTable; }document)
85 EXPORT SYLK ({aTable; }document)
167 EXPORT TEXT ({aTable; }document)

F

215 False → Boolean
253 Field (tableNum | fieldPtr{; fieldNum}) → Number | Pointer
257 Field name (fieldPtr | tableNum{; fieldNum}) → String
321 FILTER EVENT
389 FILTER KEYSTROKE (filteredChar)
230 Find in array (array; value{; start}) → Number
653 Find in field (targetField; value) → Longint
952 Find in list ({*; }list; value; scope{; itemsArray{; *}}) → Longint
449 Find window (left; top{; windowPart}) → WinRef
250 FIRST PAGE
50 FIRST RECORD {(table)}
61 FIRST SUBRECORD (subtable)
297 FLUSH BUFFERS
278 Focus object → Pointer
473 FOLDER LIST (pathname; directories)
164 FONT ({*; }object; font)
460 FONT LIST (fonts)
462 Font name (fontNumber) → String
461 Font number (fontName) → Longint
165 FONT SIZE ({*; }object; size)
166 FONT STYLE ({*; }object; styles)
388 Form event → Number
327 Frontmost process {(*)} → Integer
447 Frontmost window {(*)} → WinRef

G

691 GENERATE CERTIFICATE REQUEST (privKey; certifRequest; codeArray; nameArray)
688 GENERATE ENCRYPTION KEYPAIR (privKey; pubKey{; length})
488 Gestalt (selector; value) → Number
485 Get 4D folder ({folder}{; }{*}) → String
707 Get alignment ({*; }object) → Number
908 GET ALLOWED METHODS (methodsArray)
899 GET AUTOMATIC RELATIONS (one; many)
888 GET BACKUP INFORMATION (selector; info1; info2)
699 Get component resource ID (compName; resType; originalResNum) → Number
990 Get current data source → String
1009 Get current database localization → String
788 Get current printer → String
989 GET DATA SOURCE LIST (sourceType; sourceNamesArr; driversArr)
643 Get database parameter ({table; }selector{; stringValue}) → Longint
826 Get default user → Number
700 GET DOCUMENT ICON (docPath; icon{; size})
481 Get document position (docRef) → Number
477 GET DOCUMENT PROPERTIES (document; locked; invisible; created on; created at; modified on; modified at)
479 Get document size (document{; *}) → Number
655 Get edited text → Text
685 GET FIELD ENTRY PROPERTIES (fieldPtr|tableNum{; fieldNum}; list; mandatory; nonEnterable; nonModifiable)
258 GET FIELD PROPERTIES (fieldPtr | tableNum{; fieldNum; fieldType{; fieldLength{; indexed{; unique{; invisible}}}}})
920 GET FIELD RELATION (manyField; one; many{; *})
804 GET FIELD TITLES (table; fieldTitles; fieldNums)
976 Get file from pasteboard (xIndex) → String
898 GET FORM OBJECTS (objectsArray{; variablesArray{; pagesArray{; *}}})
969 GET FORM PARAMETER ({aTable; }form; selector; value)
674 GET FORM PROPERTIES ({table; }formName; width; height{; numPages{; fixedWidth{; fixedHeight{; title}}}})
894 Get format ({*; }object) → String
610 GET GROUP LIST (groupNames; groupNumbers)
613 GET GROUP PROPERTIES (groupID; name; owner{; members})
209 GET HIGHLIGHT (area; startSel; endSel)
902 GET HIGHLIGHTED RECORDS ({aTable; }setName)
814 GET HTTP BODY (body)

697 GET HTTP HEADER (header|fieldArray{; valueArray})
517 GET ICON RESOURCE (resID; resData{; fileRef})
510 Get indexed string (resID; strID{; resFile}) → String
255 Get last field number (tableNum | tablePtr) → Number
1045 Get Last Query Path (descFormat) → String
1046 Get Last Query Plan (descFormat) → String
1015 GET LAST SQL ERROR (codesArray; intCompArray; textArray)
254 Get last table number → Number
378 GET LIST ITEM ({*; }list; itemPos | *; itemRef; itemText{; sublist{; expanded}})
954 Get list item font ({*; }list; itemRef | *) → String
951 GET LIST ITEM ICON ({*; }list; itemRef | *; icon)
985 GET LIST ITEM PARAMETER ({*; }list; itemRef | *; selector; value)
631 GET LIST ITEM PROPERTIES ({*; }list; itemRef | *; enterable{; styles{; icon{; color}}})
632 GET LIST PROPERTIES (list; appearance{; icon{; lineHeight{; doubleClick{; multiSelections {; editable}}}}})
832 GET LISTBOX ARRAYS ({*; }object; arrColNames; arrHeaderNames; arrColVars; arrHeaderVars; arrVisible; arrStyles)
971 GET LISTBOX CELL POSITION ({*; }object; column; row{; colVar})
834 Get listbox column width ({*; }object) → Integer
917 Get listbox information ({*; }object; info) → Longint
836 Get listbox rows height ({*; }object) → Integer
1014 GET LISTBOX TABLE SOURCE ({*; }object; tableNum{; name})
991 Get localized string (resName) → String
997 GET MACRO PARAMETER (selector; textParam)
979 Get menu bar reference {(process)} → MenuRef
422 Get menu item (menu; menuItem{; process}) → String
983 GET MENU ITEM ICON (menu; menuItem; iconRef{; process})
424 Get menu item key (menu; menuItem{; process}) → Number
428 Get menu item mark (menu; menuItem{; process}) → String
981 Get menu item method (menu; menuItem{; process}) → String
980 Get menu item modifiers (menu; menuItem{; process}) → Number
972 GET MENU ITEM PROPERTY (menu; menuItem; property; value{; process})
1003 Get menu item reference (menu; menuItem) → ItemRef
426 Get menu item style (menu; menuItem; itemStyle{; process})
977 GET MENU ITEMS (menu; menuTitlesArray; menuRefsArray)
430 Get menu title (menu{; process}) → String
468 GET MOUSE (mouseX; mouseY; mouseButton{; *})
831 Get number of listbox columns ({*; }object) → Longint
915 Get number of listbox rows ({*; }object) → Longint
663 GET OBJECT RECT ({*; }object; left; top; right; bottom)

401 GET PASTEBOARD DATA (dataType; data)
958 GET PASTEBOARD DATA TYPE (4Dsignatures; nativeTypes{; formatNames})
565 GET PICTURE FROM LIBRARY (picRef | picName; picture)
522 GET PICTURE FROM PASTEBOARD (picture)
502 GET PICTURE RESOURCE (resID; resData{; resFile})
470 Get platform interface → Number
846 Get plugin access (plugIn) → String
304 Get pointer (varName) → Pointer
708 Get print marker (markNum) → Number
734 GET PRINT OPTION (option; value1{; value2})
703 GET PRINTABLE AREA (height{; width})
711 GET PRINTABLE MARGIN (left; top; right; bottom)
702 Get printed height → Number
371 GET PROCESS VARIABLE (process; srcVar; dstVar{; srcVar2; dstVar2; ...; srcVarN; dstVarN})
650 GET REGISTERED CLIENTS (clientList; methods)
686 GET RELATION PROPERTIES (fieldPtr|tableNum{; fieldNum; oneTable; oneField{; choiceField{; autoOne{; autoMany}}}})
508 GET RESOURCE (resType; resID; resData{; resFile})
513 Get resource name (resType; resID{; resFile}) → String
515 Get resource properties (resType; resID{; resFile}) → Number
889 GET RESTORE INFORMATION (selector; info1; info2)
1005 Get selected menu item reference → ItemRef
696 GET SERIAL INFORMATION (key; user; company; connected; maxUser)
909 GET SERIAL PORT MAPPING (numArray; nameArray)
784 Get SOAP info (infoNum) → String
506 Get string resource (resID{; resFile}) → String
994 GET SYSTEM FORMAT (format; value)
687 GET TABLE PROPERTIES (tablePtr|tableNum; invisible{; trigSaveNew{; trigSaveRec{; trigDelRec{; trigLoadRec}}}})
803 GET TABLE TITLES (tableTitles; tableNums)
524 Get text from pasteboard → String
504 Get text resource (resID{; resFile}) → Text
609 GET USER LIST (userNames; userNumbers)
611 GET USER PROPERTIES (userID; name; startup; password; nbLogin; lastLogin{; memberships{; groupOwner}})
683 GET WEB FORM VARIABLES (nameArray; valueArray)
780 Get Web Service error info (infoType) → String
779 GET WEB SERVICE RESULT (returnValue{; returnName{; *}})
443 GET WINDOW RECT (left; top; right; bottom{; window})
450 Get window title {(window)} → String

732 GET XML ERROR (elementRef; errorText{; row{; column}})
884 GET XSLT ERROR (errorText{; row{; column}})
206 GOTO AREA ({*; }object)
247 GOTO PAGE (pageNumber)
242 GOTO RECORD ({aTable; }record)
245 GOTO SELECTED RECORD ({table; }record)
161 GOTO XY (x; y)
169 GRAPH (graphArea; graphNumber; xLabels; yElements{; yElements2; ...; yElementsN})
298 GRAPH SETTINGS (graph; xmin; xmax; ymin; ymax; xprop; xgrid; ygrid; title{; title2; ...; titleN})
148 GRAPH TABLE ({aTable; }graphType; x field; y field{; y field2; ...; y fieldN})

H

432 HIDE MENU BAR
324 HIDE PROCESS (process)
434 HIDE TOOL BAR
436 HIDE WINDOW {(window)}
656 HIGHLIGHT RECORDS ({aTable}{; setName{; *}})
210 HIGHLIGHT TEXT (area; startSel; endSel)

I

311 IDLE
665 IMPORT DATA (fileName{; project{; *}})
86 IMPORT DIF ({aTable; }document)
87 IMPORT SYLK ({aTable; }document)
168 IMPORT TEXT ({aTable; }document)
113 In break → Boolean
191 In footer → Boolean
112 In header → Boolean
397 In transaction → Boolean
55 INPUT FORM ({aTable; }form{; userForm{; *}})
227 INSERT IN ARRAY (array; where{; howMany})
559 INSERT IN BLOB (blob; offset; len{; filler})
625 INSERT IN LIST({*; } list; beforeItemRef | *; itemText; itemRef{; expanded; sublist})
829 INSERT LISTBOX COLUMN ({*; }object; colPosition; colName; colVariable; headerName; headerVar)

970 INSERT LISTBOX COLUMN FORMULA ({*; }object; colPosition; colName; formula; dataType; headerName; headerVariable)
913 INSERT LISTBOX ROW ({*; }object; position)
412 INSERT MENU ITEM (menu; afterItem; itemText{; subMenu{; process}})
231 Insert string (source; what; where) → String
8 Int (number) → Number
548 INTEGER TO BLOB (integer; blob; byteOrder{; offset | *})
927 INTEGRATE LOG FILE (pathName)
121 INTERSECTION (set1; set2; resultSet)
93 INVERT BACKGROUND ({*; }textVar | textField)
621 Is a list (list) → Boolean
294 Is a variable (aPointer) → Boolean
492 Is compiled mode {(*)} → Boolean
716 Is data file locked → Boolean
1000 Is field number valid (tableNum | tablePtr; fieldNum) → Boolean
964 Is field value Null (aField) → Boolean
273 Is in set (set) → Boolean
714 Is license available {(license)} → Boolean
668 Is new record {(table)} → Boolean
669 Is record loaded {(table)} → Boolean
783 Is SOAP request → Boolean
999 Is table number valid (tableNum) → Boolean
616 Is user deleted (userNumber) → Boolean
520 ISO to Mac (text) → String

K

390 Keystroke → String

L

251 LAST PAGE
200 LAST RECORD {(table)}
201 LAST SUBRECORD (subtable)
811 LAUNCH EXTERNAL PROCESS (fileName{; inputStream{; outputStream{; errorStream}}})
16 Length (string) → Number
101 Level → Number
633 List item parent ({*; }list; itemRef | *) → Longint
629 List item position ({*; }list; itemRef) → Number

957 LIST OF CHOICE LISTS (numsArray; namesArray)
288 LIST TO ARRAY (list; array{; itemRefs})
556 LIST TO BLOB (list; blob{; *})
809 LIST USER FORMS (aTable; form; userFormArray)
357 LOAD COMPRESS PICTURE FROM FILE (document; method; quality; picture)
383 Load list (listName) → ListRef
52 LOAD RECORD {(table)}
185 LOAD SET ({table; }set; document)
74 LOAD VARIABLES (document; variable{; variable2; ...; variableN})
147 Locked {(table)} → Boolean
353 LOCKED ATTRIBUTES ({table; }process; 4Duser; sessionUser; processName)
22 Log (number) → Number
667 LOG EVENT (message{; importance})
928 Log File → String
647 LONGINT ARRAY FROM SELECTION (table; recordArray{; selection})
550 LONGINT TO BLOB (longInt; blob; byteOrder{; offset | *})
14 Lowercase (aString; *) → String

M

519 Mac to ISO (text) → String
463 Mac to Win (text) → String
546 Macintosh command down → Boolean
544 Macintosh control down → Boolean
545 Macintosh option down → Boolean
366 MAP FILE TYPES (macOS; windows; context)
1019 Match regex (pattern; aString{; start; pos_found; length_found; *}) → Boolean
3 Max (series) → Number
453 MAXIMIZE WINDOW {(window)}
440 Menu bar height → Longint
441 Menu bar screen → Longint
152 Menu selected {(subMenu)} → Number
88 MESSAGE (message)
175 MESSAGES OFF
181 MESSAGES ON
704 Method called on error → String
705 Method called on event → String
459 Milliseconds → Longint
4 Min (series) → Number
454 MINIMIZE WINDOW {(window)}

98 Mod (number1; number2) → Number
32 Modified (field) → Boolean
314 Modified record {(table)} → Boolean
57 MODIFY RECORD ({table}{; }{*})
204 MODIFY SELECTION ({table}{; selectMode{; enterList{; *{; *}}}})
203 MODIFY SUBRECORD (subtable; form{; *})
24 Month of (aDate) → Number
540 MOVE DOCUMENT (srcPathname; dstPathname)
664 MOVE OBJECT ({*; }object; moveH; moveV{; resizeH{; resizeV{; *}}})
844 MOVED LISTBOX COLUMN NUMBER ({*; }object; oldPosition; newPosition)
837 MOVED LISTBOX ROW NUMBER ({*; }object; oldPosition; newPosition)
718 MULTI SORT ARRAY (array{; sort}{; array2; sort2; ...; arrayN; sortN})

N

375 New list → ListRef
926 New log file → Text
317 New process (method; stack{; name{; param{; param2; ...; paramN}{; *}}}) → Number
248 NEXT PAGE
51 NEXT RECORD {(table)}
62 NEXT SUBRECORD (subtable)
448 Next window (window) → WinRef
315 Nil (aPointer) → Boolean
993 NO DEFAULT TABLE
158 NO TRACE
34 Not (boolean) → Boolean
11 Num (expression; separator) → Number

O

824 ODBC CANCEL LOAD
821 ODBC End selection → Boolean
820 ODBC EXECUTE (sqlStatement{; boundObj}{; boundObj2; ...; boundObjN})
881 ODBC EXPORT (sourceTable{; project{; *}})
825 ODBC GET LAST ERROR (errCode; errText; errODBC; errSQLServer)
819 ODBC GET OPTION (option; value)
880 ODBC IMPORT (sourceTable{; project{; *}})
822 ODBC LOAD RECORD {(numRecords)}
817 ODBC LOGIN{(dataEntry; userName; password)}

872 ODBC LOGOUT
818 ODBC SET OPTION (option; value)
823 ODBC SET PARAMETER (object; paramType)
35 Old (field) → Expression
263 OLD RELATED MANY (field)
44 OLD RELATED ONE (aField)
155 ON ERR CALL (errorMethod)
190 ON EVENT CALL (eventMethod{; processName})
189 ONE RECORD SELECT {(table)}
903 OPEN 4D PREFERENCES (selector)
312 OPEN DATA FILE (accessPath)
264 Open document (document{; fileType{; mode}}) → DocRef
309 Open external window (left; top; right; bottom; type; title; plugInArea) → Number
675 Open form window ({aTable; }formName{; type{; hPos{; vPos{; *}}}}) → WinRef
995 OPEN PRINTING JOB
497 Open resource file (resFilename{; fileType}) → DocRef
1018 OPEN SECURITY CENTER
673 OPEN WEB URL (url{; *})
153 Open window (left; top; right; bottom{; type{; title{; controlMenuBox}}}){ → WinRef }
49 ORDER BY ({aTable}{; aField{; > or <{; aField2; > or <2; ...; aFieldN; > or <N}{; *}}})
300 ORDER BY FORMULA (table{; expression{; > or <}}{; expression2; > or <2; ...; expressionN; > or <N})
107 ORDER SUBRECORDS BY (subtable; subfield{; > or <}{; subfield2; > or <2; ...; subfieldN; > or <N})
54 OUTPUT FORM ({aTable; }form{; userForm})
328 Outside call → Boolean

P

6 PAGE BREAK {(* | >)}
299 PAGE SETUP ({aTable; }form)
400 Pasteboard data size (dataType) → Number
319 PAUSE PROCESS (process)
992 PICTURE CODEC LIST (codecArray{; namesArray})
564 PICTURE LIBRARY LIST (picRefs; picNames)
457 PICTURE PROPERTIES (picture; width; height{; hOffset{; vOffset{; mode}}})
356 Picture size (picture) → Number
692 PICTURE TO BLOB (picture; pictureBlob; codec)
671 PICTURE TO GIF (pict; blobGIF)
681 PICTURE TYPE LIST (formatArray{; nameArray})

365 PLATFORM PROPERTIES (platform{; system{; processor{; language}}})
290 PLAY (objectName{; channel})
847 PLUGIN LIST (numbersArray; namesArray)
177 POP RECORD {(table)}
542 Pop up menu (contents{; default{; xCoord{; yCoord}}}) → Number
15 Position (find; aString{; start{; lengthFound{; *}}}) → Number
466 POST CLICK (mouseX; mouseY{; process}{; *})
467 POST EVENT (what; message; when; mouseX; mouseY; modifiers{; process})
465 POST KEY (code{; modifiers{; process}})
249 PREVIOUS PAGE
110 PREVIOUS RECORD {(table)}
111 PREVIOUS SUBRECORD (subtable)
5 Print form ({aTable; }form{; area1{; area2}}){ → Number }
39 PRINT LABEL ({aTable}{; document{; * | >}})
785 PRINT OPTION VALUES (option; namesArray{; info1Array{; info2Array}})
71 PRINT RECORD ({table}{; }{* | >})
60 PRINT SELECTION ({aTable}{; }{* | >})
106 PRINT SETTINGS {(dialType)}
789 PRINTERS LIST (namesArray{; altNamesArray{; modelsArray}})
275 Printing page → Number
672 Process aborted → Boolean
816 PROCESS HTML TAGS (inputData; outputData)
372 Process number (name{; *}) → Number
336 PROCESS PROPERTIES (process; procName; procState; procTime{; procVisible{; uniqueID {; origin}}})
330 Process state (process) → Number
176 PUSH RECORD {(table)}

Q

771 QR BLOB TO REPORT (area; blob)
764 QR Count columns (area) → Longint
749 QR DELETE COLUMN (area; colNumber)
754 QR DELETE OFFSCREEN AREA (area)
791 QR EXECUTE COMMAND (area; command)
776 QR Find column (area; expression) → Longint
795 QR Get area property (area; property) → Longint
798 QR GET BORDERS (area; column; row; border; line{; color})
792 QR Get command status (area; command{; value}) → Longint
756 QR GET DESTINATION (area; type{; specifics})

773 QR Get document property (area; property) → Longint
747 QR Get drop column (area) → Longint
775 QR GET HEADER AND FOOTER (area; selector; leftTitle; centerTitle; rightTitle; height{; picture{; pictAlignment}})
751 QR Get HTML template (area) → Text
766 QR GET INFO COLUMN (area; colNum; title; object; hide; size; repeatedValue; displayFormat)
769 QR Get info row (area; row) → Longint
755 QR Get report kind (area) → Longint
758 QR Get report table (area) → Longint
793 QR GET SELECTION (area; left; top{; right{; bottom}})
753 QR GET SORTS (area; aColumns{; aOrders})
760 QR Get text property (area; colNum; rowNum; property) → Longint
768 QR GET TOTALS DATA (area; colNum; breakNum; operator; text)
762 QR GET TOTALS SPACING (area; subtotal; value)
748 QR INSERT COLUMN (area; colNumber; object)
735 QR New offscreen area → Longint
790 QR ON COMMAND (area; methodName)
197 QR REPORT ({aTable; }document{; hierarchical{; wizard{; search{; *}}}})
770 QR REPORT TO BLOB (area; blob)
746 QR RUN (area)
796 QR SET AREA PROPERTY (area; property; value)
797 QR SET BORDERS (area; column; row; border; line{; color})
745 QR SET DESTINATION (area; type; specifics)
772 QR SET DOCUMENT PROPERTY (area; property; value)
774 QR SET HEADER AND FOOTER (area; selector; leftTitle; centerTitle; rightTitle; height {; picture{; pictAlignment}})
750 QR SET HTML TEMPLATE (area; template)
765 QR SET INFO COLUMN (area; colNum; title; object; hide; size; repeatedValue; displayFormat)
763 QR SET INFO ROW (area; row; hide)
738 QR SET REPORT KIND (area; type)
757 QR SET REPORT TABLE (area; table)
794 QR SET SELECTION (area; left; top; right; bottom)
752 QR SET SORTS (area; aColumns{; aOrders})
759 QR SET TEXT PROPERTY (area; colNum; rowNum; property; value)
767 QR SET TOTALS DATA (area; colNum; breakNum; operator | value)
761 QR SET TOTALS SPACING (area; subtotal; value)
277 QUERY ({aTable}{; queryArgument{; *}})
292 QUERY BY EXAMPLE ({aTable}{; }{*})

48 QUERY BY FORMULA ({aTable}{; }{queryFormula})
942 QUERY BY SQL ({aTable; }sqlFormula)
341 QUERY SELECTION ({aTable}{; queryArgument{; *}})
207 QUERY SELECTION BY FORMULA (aTable{; queryFormula})
108 QUERY SUBRECORDS (subtable; queryFormula)
644 QUERY WITH ARRAY (targetField; array)
291 QUIT 4D {(time)}

R

100 Random → Number
145 READ ONLY {(table | *)}
362 Read only state {(table)} → Boolean
678 READ PICTURE FILE (fileName; picture)
146 READ WRITE {(table | *)}
552 REAL TO BLOB (real; blob; realFormat{; offset | *})
172 RECEIVE BUFFER (receiveVar)
104 RECEIVE PACKET ({docRef; }receiveVar; stopChar | numChars)
79 RECEIVE RECORD {(table)}
81 RECEIVE VARIABLE (variable)
243 Record number {(table)} → Number
76 Records in selection {(table)} → Number
195 Records in set (set) → Number
7 Records in subselection (subtable) → Number
83 Records in table {(table)} → Number
174 REDRAW (object)
382 REDRAW LIST (list)
456 REDRAW WINDOW {(window)}
351 REDUCE SELECTION ({aTable; }number)
648 REGISTER CLIENT (clientName{; period{; *}})
38 REJECT {(aField)}
262 RELATE MANY (oneTable | Field)
340 RELATE MANY SELECTION (aField)
42 RELATE ONE (manyTable | Field{; choiceField})
349 RELATE ONE SELECTION (manyTable; oneTable)
978 RELEASE MENU (menu)
561 REMOVE FROM SET ({table; }set)
567 REMOVE PICTURE FROM LIBRARY (picRef | picName)
233 Replace string (source; oldString; newString{; howMany}{; *}) → String
163 Request (message{; defaultResponse{; OKButtonTitle{; CancelButtonTitle}}}) → String

890 RESIZE FORM WINDOW (width; height)
695 RESOLVE ALIAS (aliasPath; targetPath)
394 RESOLVE POINTER (pointer; varName; tableNum; fieldNum)
500 RESOURCE LIST (resType; resIDs; resNames{; resFile})
499 RESOURCE TYPE LIST (resTypes{; resFile})
918 RESTORE
320 RESUME PROCESS (process)
712 Right click → Boolean
94 Round (round; places) → Number

S

384 SAVE LIST (list; listName)
358 SAVE PICTURE TO FILE (document; picture)
53 SAVE RECORD {(table)}
43 SAVE RELATED ONE (aField)
184 SAVE SET (set; document)
75 SAVE VARIABLES (document; variable{; variable2; ...; variableN})
857 SAX ADD PROCESSING INSTRUCTION (document; statement)
856 SAX ADD XML CDATA (document; data)
852 SAX ADD XML COMMENT (document; comment)
851 SAX ADD XML DOCTYPE (document; docType)
855 SAX ADD XML ELEMENT VALUE (document; data; *)
854 SAX CLOSE XML ELEMENT (document)
878 SAX GET XML CDATA (document; value)
874 SAX GET XML COMMENT (document; comment)
873 SAX GET XML DOCUMENT VALUES (document; encoding; version; standalone)
876 SAX GET XML ELEMENT (document; name; prefix; attrNames; attrValues)
877 SAX GET XML ELEMENT VALUE (document; value)
879 SAX GET XML ENTITY (document; name; value)
860 SAX Get XML node (document) → Longint
875 SAX GET XML PROCESSING INSTRUCTION (document; name; value)
853 SAX OPEN XML ELEMENT (document; tag{; attribName{; attribValue}}{; attribName2; attribValue2; ...; attribNameN; attribValueN})
921 SAX OPEN XML ELEMENT ARRAYS (document; tag{; attribNamesArray {; attribValuesArray}}{; attribNamesArray2; attribValuesArray2; ...; attribNamesArrayN; attribValuesArrayN})
858 SAX SET XML OPTIONS (document; encoding{; standalone{; indentation}})
350 SCAN INDEX (aField; number{; > or <})
438 SCREEN COORDINATES (left; top; right; bottom{; screen})

439 SCREEN DEPTH (depth; color{; screen})
188 Screen height {(*)} → Number
187 Screen width {(*)} → Number
906 SCROLL LINES ({*; }object{; position{; *}})
698 Secured Web connection → Boolean
905 Select document (directory; fileTypes; title; options{; selected}) → String
670 Select folder ({message}{; }{defaultPath}) → String
381 SELECT LIST ITEMS BY POSITION ({*; }list; itemPos{; positionsArray})
630 SELECT LIST ITEMS BY REFERENCE (list; itemRef{; refArray})
912 SELECT LISTBOX ROW ({*; }object; position{; action})
345 SELECT LOG FILE (logFile | *)
956 Select RGB Color ({defaultColor}{; }{message}) → Longint
379 Selected list items ({*; }list{; itemsArray{; *}}) → Longint
246 Selected record number {(table)} → Number
368 SELECTION RANGE TO ARRAY (start; end; field | table; array{; field2 | table2; array2; ...; fieldN | tableN; arrayN})
260 SELECTION TO ARRAY (field | table; array{; field2 | table2; array2; ...; fieldN | tableN; arrayN})
308 Self → Pointer
143 Semaphore (semaphore{; tickCount}) → Boolean
654 SEND HTML BLOB (blob; type{; noContext})
619 SEND HTML FILE (htmlFile)
677 SEND HTML TEXT (htmlText{; noContext})
815 SEND HTTP RAW DATA (data{; *})
659 SEND HTTP REDIRECT (url{; *})
103 SEND PACKET ({docRef; }packet)
78 SEND RECORD {(table)}
781 SEND SOAP FAULT (faultType; description)
80 SEND VARIABLE (variable)
244 Sequence number {(aTable)} → Number
316 SET ABOUT (itemText; method)
706 SET ALIGNMENT ({*; }object; alignment)
805 SET ALLOWED METHODS (methodsArray)
310 SET AUTOMATIC RELATIONS (one{; many})
606 SET BLOB SIZE (blob; size{; filler})
813 SET CGI EXECUTABLE (url1{; url2})
77 SET CHANNEL (port | operation{; settings | document})
237 SET CHOICE LIST ({*; }object; list)
271 SET COLOR ({*; }object; color{; altColor})
787 SET CURRENT PRINTER (printerName)

469 SET CURSOR {(cursor)}
642 SET DATABASE PARAMETER ({aTable; }selector; value)
392 SET DEFAULT CENTURY (century{; pivotYear})
904 SET DICTIONARY (dictionary)
531 SET DOCUMENT CREATOR (document; fileCreator)
482 SET DOCUMENT POSITION (docRef; offset{; anchor})
478 SET DOCUMENT PROPERTIES (document; locked; invisible; created on; created at; modified on; modified at)
480 SET DOCUMENT SIZE (document; size)
530 SET DOCUMENT TYPE (document; fileType)
238 SET ENTERABLE ({*; }entryArea; enterable)
812 SET ENVIRONMENT VARIABLE (varName; varValue)
919 SET FIELD RELATION (manyTable | manyField; one; many)
602 SET FIELD TITLES (aTable | aSubtable; fieldTitles; fieldNumbers{; *})
965 SET FIELD VALUE NULL (aField)
975 SET FILE TO PASTEBOARD (filePath)
235 SET FILTER ({*; }object; entryFilter)
892 SET FORM HORIZONTAL RESIZING (resize{; minWidth{; maxWidth}})
891 SET FORM SIZE ({object; }horizontal; vertical{; *})
893 SET FORM VERTICAL RESIZING (resize{; minHeight{; maxHeight}})
236 SET FORMAT ({*; }object; displayFormat)
614 Set group properties (groupID; name; owner{; members}) → Number
639 SET HOME PAGE (homePage)
634 SET HTML ROOT (rootFolder)
660 SET HTTP HEADER (header|fieldArray{; valueArray})
344 SET INDEX (aField; index{; mode{; *}})
385 SET LIST ITEM({*; } list; itemRef | *; newItemText; newItemRef{; sublist; expanded})
953 SET LIST ITEM FONT ({*; }list; itemRef | *; font)
950 SET LIST ITEM ICON ({*; }list; itemRef | *; icon)
986 SET LIST ITEM PARAMETER ({*; }list; itemRef | *; selector; value)
386 SET LIST ITEM PROPERTIES ({*; }list; itemRef | *; enterable; styles; icon{; color})
387 SET LIST PROPERTIES (list; appearance{; icon{; lineHeight{; doubleClick{; multiSelections {; editable}}}}})
833 SET LISTBOX COLUMN WIDTH ({*; }object; width)
842 SET LISTBOX GRID COLOR ({*; }object; color; horizontal; vertical)
835 SET LISTBOX ROWS HEIGHT ({*; }object; height)
1013 SET LISTBOX TABLE SOURCE ({*; }object; tableNum | name)
998 SET MACRO PARAMETER (selector; textParam)
67 SET MENU BAR (menuBar{; process{; *}})
348 SET MENU ITEM (menu; menuItem; itemText{; process})

984 SET MENU ITEM ICON (menu; menuItem; iconRef{; process})
208 SET MENU ITEM MARK (menu; menuItem; mark{; process})
982 SET MENU ITEM METHOD (menu; menuItem; methodName{; process})
973 SET MENU ITEM PROPERTY (menu; menuItem; property; value{; process})
1004 SET MENU ITEM REFERENCE (menu; menuItem; itemRef)
423 SET MENU ITEM SHORTCUT (menu; menuItem; itemKey{; modifiers{; process}})
425 SET MENU ITEM STYLE (menu; menuItem; itemStyle{; process})
503 SET PICTURE RESOURCE (resID; resData{; resFile})
566 SET PICTURE TO LIBRARY (picture; picRef; picName)
521 SET PICTURE TO PASTEBOARD (picture)
367 SET PLATFORM INTERFACE (interface)
845 SET PLUGIN ACCESS (plugIn; group)
709 SET PRINT MARKER (markNum; position{; *})
733 SET PRINT OPTION (option; value1{; value2})
364 SET PRINT PREVIEW (preview)
710 SET PRINTABLE MARGIN (left; top; right; bottom)
370 SET PROCESS VARIABLE (process; dstVar; expr{; dstVar2; expr2; ...; dstVarN; exprN})
661 SET QUERY AND LOCK (lock)
396 SET QUERY DESTINATION (destinationType{; destinationObject})
395 SET QUERY LIMIT (limit)
623 SET REAL COMPARISON LEVEL (epsilon)
509 SET RESOURCE (resType; resID; resData{; resFile})
514 SET RESOURCE NAME (resType; resID; resName{; resFile})
516 SET RESOURCE PROPERTIES (resType; resID; resAttr{; resFile})
628 SET RGB COLORS ({*; }object; foregroundColor; backgroundColor{; altBackgrndColor})
537 SET SCREEN DEPTH (depth{; color{; screen}})
843 SET SCROLLBAR VISIBLE ({*; }object; horizontal; vertical)
507 SET STRING RESOURCE (resID; resData{; resFile})
601 SET TABLE TITLES (tableTitles; tableNumbers{; *})
505 SET TEXT RESOURCE (resID; resData{; resFile})
523 SET TEXT TO PASTEBOARD (text)
268 SET TIMEOUT (seconds)
645 SET TIMER (tickCount)
612 Set user properties (userID; name; startup; password; nbLogin; lastLogin{; memberships {; groupOwner}}) → Number
603 SET VISIBLE ({*; }object; visible)
620 SET WEB DISPLAY LIMITS (numberRecords{; numberPages{; picRef}})
901 SET WEB SERVICE OPTION (option; value)
777 SET WEB SERVICE PARAMETER (name; value{; soapType})
622 SET WEB TIMEOUT (timeout)

444 SET WINDOW RECT (left; top; right; bottom{; window})
213 SET WINDOW TITLE (title{; window})
883 SET XSLT PARAMETER (paramName; paramValue)
543 Shift down → Boolean
841 SHOW LISTBOX GRID ({*; }object; horizontal; vertical)
431 SHOW MENU BAR
922 SHOW ON DISK (pathname{; *})
325 SHOW PROCESS (process)
433 SHOW TOOL BAR
435 SHOW WINDOW {(window)}
17 Sin (number) → Number
274 Size of array (array) → Number
782 SOAP DECLARATION (variable; type; input_output{; alias})
229 SORT ARRAY (array{; array2; ...; arrayN}{; > or <})
391 SORT LIST (list{; > or <})
916 SORT LISTBOX COLUMNS ({*; }object; colNum; order{; colNum2; order2; ...; colNumN; orderN})
900 SPELL CHECKING
539 Square root (number) → Number
962 START SQL SERVER
239 START TRANSACTION
617 START WEB SERVER
26 Std deviation (series) → Number
963 STOP SQL SERVER
618 STOP WEB SERVER
10 String (expression{; format}) → String
511 STRING LIST TO ARRAY (resID; strings{; resFile})
489 Structure file {(*)} → String
12 Substring (source; firstChar{; numChars}) → String
97 Subtotal (data{; pageBreak}) → Number
1 Sum (series) → Number
28 Sum squares (series) → Number
487 System folder {(type)} → String

T

252 Table (tableNum | aPtr) → Pointer | Number
256 Table name (tableNum | tablePtr) → String
19 Tan (number) → Number
486 Temporary folder → String

476 Test path name (pathname) → Number
652 Test semaphore (semaphore) → Boolean
554 TEXT TO BLOB (text; blob; textFormat{; offset | *})
458 Tickcount → Number
179 Time (timeString) → Time
180 Time string (seconds) → String
1016 Tool bar height → Longint
157 TRACE
961 Transaction level → Longint
988 TRANSFORM PICTURE (picture; operator{; param1{; param2{; param3{; param4}}}})
398 Trigger level → Number
399 TRIGGER PROPERTIES (triggerLevel; dbEvent; tableNum; recordNum)
214 True → Boolean
95 Trunc (number; places) → Number
295 Type (fieldVar) → Number

U

82 Undefined (variable) → Boolean
120 UNION (set1; set2; resultSet)
212 UNLOAD RECORD {(table)}
649 UNREGISTER CLIENT
13 Uppercase (aString; *) → String
205 USE CHARACTER SET (map | *{; mapInOut})
959 USE EXTERNAL DATABASE (sourceName{; user; password})
960 USE INTERNAL DATABASE
332 USE NAMED SELECTION (name)
118 USE SET (set)
338 User in group (user; group) → Boolean
849 USERS TO BLOB (users)

V

946 Validate Digest Web Password (userName; password) → Boolean
638 Validate password (userID; password) → Boolean
240 VALIDATE TRANSACTION
532 VARIABLE TO BLOB (variable; blob{; offset | *})
635 VARIABLE TO VARIABLE (process; dstVar; srcVar{; dstVar2; srcVar2; ...; dstVarN; srcVarN})
27 Variance (series) → Number

1008 VERIFY CURRENT DATA FILE (objects; options; method{; tablesArray; fieldsArray})
939 VERIFY DATA FILE (structurePath; dataPath; objects; options; method{; tablesArray; fieldsArray})
495 Version type → Long Integer
472 VOLUME ATTRIBUTES (volume; size; used; free)
471 VOLUME LIST (volumes)

W

658 WEB CACHE STATISTICS (pages; hits; usage)
657 Web Context → Boolean
464 Win to Mac (text) → String
445 Window kind {(window)}
442 WINDOW LIST (windows{; *})
446 Window process {(window)} → Number
563 Windows Alt down → Boolean
562 Windows Ctrl down → Boolean
680 WRITE PICTURE FILE (fileName; picture{; codec})

Y

25 Year of (date) → Number

Constants

4D Environment

Related command(s): Application type, Get 4D folder, GET MACRO PARAMETER, SET MACRO PARAMETER, Version type.

Constant	Type	Value
4D Client	Long Integer	4
4D Client Database Folder	Long Integer	3
4D Desktop	Long Integer	3
4D Developer	Long Integer	0
4D First	Long Integer	6
4D Interpreted Desktop	Long Integer	2
4D Server	Long Integer	5
4D Unlimited Desktop	Long Integer	1
Active 4D Folder	Long Integer	0
Current Resources Folder	Long Integer	6
Database Folder	Long Integer	4
Database Folder Unix Syntax	Long Integer	5
Demo Version	Long Integer	1
Extras Folder	Long Integer	2
Full method text	Long Integer	1
Full Version	Long Integer	0
Highlighted method text	Long Integer	2
Licenses Folder	Long Integer	1

ASCII Codes

Related command(s): Char, ON EVENT CALL.

Constant	Type	Value
ACK ASCII code	Long Integer	6
At sign	Long Integer	64
Backspace	Long Integer	8
BEL ASCII code	Long Integer	7
BS ASCII code	Long Integer	8
CAN ASCII code	Long Integer	24
Carriage return	Long Integer	13
CR ASCII code	Long Integer	13
DC1 ASCII code	Long Integer	17
DC2 ASCII code	Long Integer	18
DC3 ASCII code	Long Integer	19
DC4 ASCII code	Long Integer	20
DEL ASCII code	Long Integer	127
DLE ASCII code	Long Integer	16
Double quote	Long Integer	34
EM ASCII code	Long Integer	25
ENQ ASCII code	Long Integer	5
Enter	Long Integer	3
EOT ASCII code	Long Integer	4
ESC ASCII code	Long Integer	27
Escape	Long Integer	27
ETB ASCII code	Long Integer	23
ETX ASCII code	Long Integer	3
FF ASCII code	Long Integer	12
FS ASCII code	Long Integer	28
GS ASCII code	Long Integer	29
HT ASCII code	Long Integer	9
LF ASCII code	Long Integer	10
Line feed	Long Integer	10
NAK ASCII code	Long Integer	21
NBSP	Long Integer	202
NUL ASCII code	Long Integer	0
Period	Long Integer	46
Quote	Long Integer	39
RS ASCII code	Long Integer	30
SI ASCII code	Long Integer	15

ASCII Codes (continued)

Constant	Type	Value
SO ASCII code	Long Integer	14
SOH ASCII code	Long Integer	1
SP ASCII code	Long Integer	32
Space	Long Integer	32
STX ASCII code	Long Integer	2
SUB ASCII code	Long Integer	26
SYN ASCII code	Long Integer	22
Tab	Long Integer	9
US ASCII code	Long Integer	31
VT ASCII code	Long Integer	11

Backup and Restore

Related command(s): GET BACKUP INFORMATION, GET RESTORE INFORMATION.

Constant	Type	Value
Last Backup Date	Long Integer	0
Last Backup Status	Long Integer	2
Last Restore Date	Long Integer	0
Last Restore Status	Long Integer	2
Next Backup Date	Long Integer	4

BLOB

Related command(s): BLOB PROPERTIES, BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

Constant	Type	Value
Compact compression mode	Long Integer	1
Extended real format	Long Integer	1
Fast compression mode	Long Integer	2
Is not compressed	Long Integer	0
Mac C string	Long Integer	0
Mac Pascal string	Long Integer	1
Mac Text with length	Long Integer	2
Mac Text without length	Long Integer	3
Macintosh byte ordering	Long Integer	1
Macintosh double real format	Long Integer	2
Native byte ordering	Long Integer	0
Native real format	Long Integer	0
PC byte ordering	Long Integer	2
PC double real format	Long Integer	3
UTF8 C string	Long Integer	4
UTF8 Text with length	Long Integer	5
UTF8 Text without length	Long Integer	6

Colors

Related command(s): SET COLOR.

Constant	Type	Value
Black	Long Integer	15
Blue	Long Integer	6
Brown	Long Integer	13
Dark Blue	Long Integer	5
Dark Brown	Long Integer	10
Dark Green	Long Integer	9
Dark Grey	Long Integer	11
Green	Long Integer	8
Grey	Long Integer	14
Light Blue	Long Integer	7
Light Grey	Long Integer	12
Orange	Long Integer	2
Purple	Long Integer	4
Red	Long Integer	3
White	Long Integer	0
Yellow	Long Integer	1

Communications

Related command(s): SET CHANNEL.

Constant	Type	Value
Data bits 5	Long Integer	0
Data bits 6	Long Integer	2048
Data bits 7	Long Integer	1024
Data bits 8	Long Integer	3072
MacOS Printer Port	Long Integer	0
MacOS Serial Port	Long Integer	1
Parity Even	Long Integer	12288
Parity None	Long Integer	0
Parity Odd	Long Integer	4096
Protocol DTR	Long Integer	30
Protocol None	Long Integer	0
Protocol XONXOFF	Long Integer	20
Speed 115200	Long Integer	1022
Speed 1200	Long Integer	94
Speed 1800	Long Integer	62
Speed 19200	Long Integer	4
Speed 230400	Long Integer	1021
Speed 2400	Long Integer	46
Speed 300	Long Integer	380
Speed 3600	Long Integer	30
Speed 4800	Long Integer	22
Speed 57600	Long Integer	0
Speed 600	Long Integer	189
Speed 7200	Long Integer	14
Speed 9600	Long Integer	10
Stop bits One	Long Integer	16384
Stop bits One and a half	Long Integer	-32768
Stop bits Two	Long Integer	-16384

Data file maintenance

Constant	Type	Value
Create process	Long Integer	32768
Do not compact index	Long Integer	2
Do not create log file	Long Integer	16384
Move to Replaced files folder		4
New file	Long Integer	0
New file dialog	Long Integer	1
Renumber records	Long Integer	1
Use selected file	Long Integer	2
Verify All No Callback	Long Integer	16
Verify Indexes	Long Integer	8
Verify Records	Long Integer	4

Database Engine

Related command(s): Record number.

Constant	Type	Value
New record	Long Integer	-3
No current record	Long Integer	-1

Database Events

Related command(s): Database event, TRIGGER PROPERTIES.

Constant	Type	Value
On Deleting Record Event	Long Integer	3
On Loading Record Event	Long Integer	4
On Saving Existing Record Event	Long Integer	2
On Saving New Record Event	Long Integer	1

Database Parameters

Related command(s): Get database parameter, SET DATABASE PARAMETER.

Constant	Type	Value
4D Client Scheduler	Long Integer	12
4D Client Timeout	Long Integer	14
4D Developer Scheduler	Long Integer	10
4D Server Log Recording	Long Integer	28
4D Server Scheduler	Long Integer	11
4D Server Timeout	Long Integer	13
Cache writing mode	Long Integer	26
Character set	Long Integer	17
Client Character set	Long Integer	24
Client HTTPS Port ID	Long Integer	40
Client IP Address to listen	Long Integer	23
Client Max Concurrent Web Proc	Long Integer	25
Client Max Web requests size	Long Integer	21
Client Maximum Web Process	Long Integer	20
Client Minimum Web Process	Long Integer	19
Client Port ID	Long Integer	22
Client Server Port ID	Long Integer	35
Client Web Log Recording	Long Integer	30
Database Cache Size	Long Integer	9
Debug Log Recording	Long Integer	34
HTTPS Port ID	Long Integer	39
Index Compacting	Long Integer	4
Invert Objects	Long Integer	37
IP Address to listen	Long Integer	16
Max Concurrent Web Processes	Long Integer	18
Maximum Web Process	Long Integer	7
Maximum Web requests size	Long Integer	27
Minimum Web Process	Long Integer	6
Port ID	Long Integer	15
Real Display Precision	Long Integer	32
Seq Access Optimization	Long Integer	2
Seq Distinct Values Ratio	Long Integer	3
Seq Order Ratio	Long Integer	1
Seq Query Select Ratio	Long Integer	5
SQL Autocommit	Long Integer	43
SQL Engine Case Sensitivity	Long Integer	44

Database Parameters (continued)

Constant	Type	Value
Table Sequence Number	Long Integer	31
TCP_NODELAY	Long Integer	33
Temporary memory size	Long Integer	42
Unicode mode	Long Integer	41
Web Conversion Mode	Long Integer	8
Web Log Recording	Long Integer	29
WEDD Signature	Long Integer	36

Date Display Formats

Related command(s): Get format, SET FORMAT, String.

Constant	Type	Value
Blank if null date	Long Integer	100
Internal date abbreviated	Long Integer	6
Internal date long	Long Integer	5
Internal date short	Long Integer	7
Internal date short special	Long Integer	4
ISO Date	Long Integer	8
System date abbreviated	Long Integer	2
System date long	Long Integer	3
System date short	Long Integer	1

Days and Months

Related command(s): Day number, Month of.

Constant	Type	Value
April	Long Integer	4
August	Long Integer	8
December	Long Integer	12
February	Long Integer	2
Friday	Long Integer	6
January	Long Integer	1
July	Long Integer	7
June	Long Integer	6
March	Long Integer	3
May	Long Integer	5
Monday	Long Integer	2
November	Long Integer	11
October	Long Integer	10
Saturday	Long Integer	7
September	Long Integer	9
Sunday	Long Integer	1
Thursday	Long Integer	5
Tuesday	Long Integer	3
Wednesday	Long Integer	4

Dictionaries

Related command(s): SET DICTIONARY.

Constant	Type	Value
English Dictionary	Long Integer	69632
French Dictionary	Long Integer	262144
German Dictionary	Long Integer	131584
Norwegian Dictionary	Long Integer	589824
Spanish Dictionary	Long Integer	196608

Euro currencies

Related command(s): Euro converter.

Constant	Type	Value
Austrian Schilling	String	ATS
Belgian Franc	String	BEF
Deutschemark	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)

Related command(s): POST EVENT, POST KEY, SET MENU ITEM SHORTCUT.

Constant	Type	Value
Activate window bit	Long Integer	0
Activate window mask	Long Integer	1
Caps Lock key bit	Long Integer	10
Caps Lock key mask	Long Integer	1024
Command key bit	Long Integer	8
Command key mask	Long Integer	256
Control key bit	Long Integer	12
Control key mask	Long Integer	4096
Mouse button bit	Long Integer	7
Mouse button mask	Long Integer	128
Option key bit	Long Integer	11
Option key mask	Long Integer	2048
Right control key bit	Long Integer	15
Right control key mask	Long Integer	32768
Right option key bit	Long Integer	14
Right option key mask	Long Integer	16384
Right shift key bit	Long Integer	13
Right shift key mask	Long Integer	8192
Shift key bit	Long Integer	9
Shift key mask	Long Integer	512

Events (What)

Related command(s): POST EVENT.

Constant	Type	Value
Activate event	Long Integer	8
Auto key event	Long Integer	5
Disk event	Long Integer	7
Key down event	Long Integer	3
Key up event	Long Integer	4
Mouse down event	Long Integer	1
Mouse up event	Long Integer	2
Null event	Long Integer	0
Operating system event	Long Integer	15
Update event	Long Integer	6

Expressions

Related command(s): RECEIVE BUFFER, RECEIVE PACKET.

Constant	Type	Value
MAXINT	Long Integer	32767
MAXLONG	Long Integer	2147483647
MAXTEXTLENBEFOREV11	Long Integer	32000

External data source

Related command(s): GET DATA SOURCE LIST, ODBC GET OPTION, ODBC LOAD RECORD, ODBC LOGIN, ODBC SET OPTION, ODBC SET PARAMETER.

Constant	Type	Value
ODBC All Records	Long Integer	-1
ODBC Asynchronous	Long Integer	1
ODBC Connection Timeout	Long Integer	5
ODBC Max Data Length	Long Integer	3
ODBC Max Rows	Long Integer	2
ODBC Param In	Long Integer	1
ODBC Param In Out	Long Integer	2
ODBC Param Out	Long Integer	4
ODBC Query Timeout	Long Integer	4
SQL_INTERNAL	String	;DB4D_SQL_LOCAL;
System Data Source	Long Integer	2
User Data Source	Long Integer	1

Field and Variable Types

Related command(s): GET FIELD PROPERTIES, SOAP DECLARATION, Type.

Constant	Type	Value
Array 2D	Long Integer	13
Boolean array	Long Integer	22
Date array	Long Integer	17
Integer array	Long Integer	15
Is Alpha Field	Long Integer	0
Is BLOB	Long Integer	30
Is Boolean	Long Integer	6
Is Date	Long Integer	4
Is Integer	Long Integer	8
Is LongInt	Long Integer	9
Is Picture	Long Integer	3
Is Pointer	Long Integer	23
Is Real	Long Integer	1
Is String Var	Long Integer	24
Is Subtable	Long Integer	7
Is Text	Long Integer	2
Is Time	Long Integer	11
Is Undefined	Long Integer	5
LongInt array	Long Integer	16
Picture array	Long Integer	19
Pointer array	Long Integer	20
Real array	Long Integer	14
String array	Long Integer	21
Text array	Long Integer	18

Find window

Related command(s): Find window.

Constant	Type	Value
In contents	Long Integer	3
In drag	Long Integer	4
In go away	Long Integer	6
In grow	Long Integer	5
In menu bar	Long Integer	1
In system window	Long Integer	2
In zoom box	Long Integer	8

Font Styles

Related command(s): FONT STYLE, Get menu item style, SET LIST ITEM PROPERTIES, SET MENU ITEM STYLE.

Constant	Type	Value
Bold	Long Integer	1
Condensed	Long Integer	32
Extended	Long Integer	64
Italic	Long Integer	2
Outline	Long Integer	8
Plain	Long Integer	0
Shadow	Long Integer	16
Underline	Long Integer	4

Form area

Related command(s): Get print marker, Print form, SET PRINT MARKER.

Constant	Type	Value
Form Break0	Long Integer	300
Form Break1	Long Integer	301
Form Break2	Long Integer	302
Form Break3	Long Integer	303
Form Break4	Long Integer	304
Form Break5	Long Integer	305
Form Break6	Long Integer	306
Form Break7	Long Integer	307
Form Break8	Long Integer	308
Form Break9	Long Integer	309
Form Detail	Long Integer	0
Form Footer	Long Integer	100
Form Header	Long Integer	200
Form Header1	Long Integer	201
Form Header10	Long Integer	210
Form Header2	Long Integer	202
Form Header3	Long Integer	203
Form Header4	Long Integer	204
Form Header5	Long Integer	205
Form Header6	Long Integer	206
Form Header7	Long Integer	207
Form Header8	Long Integer	208
Form Header9	Long Integer	209

Form Events

Related command(s): Form event.

Constant	Type	Value
On Activate	Long Integer	11
On After Edit	Long Integer	45
On After Keystroke	Long Integer	28
On After Sort	Long Integer	30
On Arrow Click	Long Integer	38
On Before Data Entry	Long Integer	41
On Before Keystroke	Long Integer	17
On Begin Drag Over	Long Integer	46
On Clicked	Long Integer	4
On Close Box	Long Integer	22
On Close Detail	Long Integer	26
On Collapse	Long Integer	44
On Column Moved	Long Integer	32
On Column Resize	Long Integer	33
On Data Change	Long Integer	20
On Deactivate	Long Integer	12
On Display Detail	Long Integer	8
On Double Clicked	Long Integer	13
On Drag Over	Long Integer	21
On Drop	Long Integer	16
On Expand	Long Integer	43
On Getting Focus	Long Integer	15
On Header	Long Integer	5
On Header Click	Long Integer	42
On Load	Long Integer	1
On Load Record	Long Integer	40
On Long Click	Long Integer	39
On Losing Focus	Long Integer	14
On Menu Selected	Long Integer	18
On Mouse Enter	Long Integer	35
On Mouse Leave	Long Integer	36
On Mouse Move	Long Integer	37
On Open Detail	Long Integer	25
On Outside Call	Long Integer	10
On Plug in Area	Long Integer	19
On Printing Break	Long Integer	6

Form Events (continued)

Constant	Type	Value
On Printing Detail	Long Integer	23
On Printing Footer	Long Integer	7
On Resize	Long Integer	29
On Row Moved	Long Integer	34
On Selection Change	Long Integer	31
On Timer	Long Integer	27
On Unload	Long Integer	24
On Validate	Long Integer	3

Form Parameters

Related command(s): DISPLAY SELECTION, GET FORM PARAMETER, MODIFY SELECTION.

Constant	Type	Value
Multiple Selection	Long Integer	2
No Selection	Long Integer	0
NonInverted Objects	Long Integer	0
Single Selection	Long Integer	1

Function Keys

Related command(s): ON EVENT CALL.

Constant	Type	Value
Backspace Key	Long Integer	8
Down Arrow Key	Long Integer	31
End Key	Long Integer	4
Enter Key	Long Integer	3
Escape Key	Long Integer	27
F1 Key	Long Integer	-122
F10 Key	Long Integer	-109
F11 Key	Long Integer	-103
F12 Key	Long Integer	-111
F13 Key	Long Integer	-105
F14 Key	Long Integer	-107
F15 Key	Long Integer	-113
F2 Key	Long Integer	-120
F3 Key	Long Integer	-99
F4 Key	Long Integer	-118
F5 Key	Long Integer	-96
F6 Key	Long Integer	-97
F7 Key	Long Integer	-98
F8 Key	Long Integer	-100
F9 Key	Long Integer	-101
Help Key	Long Integer	5
Home Key	Long Integer	1
Left Arrow Key	Long Integer	28
Page Down Key	Long Integer	12
Page Up Key	Long Integer	11
Return Key	Long Integer	13
Right Arrow Key	Long Integer	29
Tab Key	Long Integer	9
Up Arrow Key	Long Integer	30

Hierarchical Lists

Related command(s): GET LIST PROPERTIES, SET LIST PROPERTIES.

Constant	Type	Value
Additional text	String	4D_additional_text
Ala Macintosh	Long Integer	1
Ala Windows	Long Integer	2
Macintosh node	Long Integer	860
Use PicRef	Long Integer	131072
Use PICT resource	Long Integer	65536
Windows node	Long Integer	138

Index Type

Related command(s): CREATE INDEX, SET INDEX.

Constant	Type	Value
Cluster BTree Index	Long Integer	3
Default Index Type	Long Integer	0
Keywords Index	Long Integer	-1
Standard BTree Index	Long Integer	1

Is license available

Related command(s): Get plugin access, Is license available, PLUGIN LIST, SET PLUGIN ACCESS.

Constant	Type	Value
4D Client SOAP License	Long Integer	808465465
4D Client Web License	Long Integer	808465209
4D Draw License	Long Integer	808464694
4D for ADO License	Long Integer	808465714
4D for MySQL License	Long Integer	808465712
4D for OCI License	Long Integer	808465208
4D for PostgreSQL License	Long Integer	808465713
4D for Sybase License	Long Integer	808465715
4D ODBC Pro License	Long Integer	808464946
4D SOAP License	Long Integer	808465464
4D SOAP Local License	Long Integer	808531000
4D SOAP One Connection Licence	Long Integer	825242680
4D SQL Server License	Long Integer	808464949
4D SQL Server Local License		808530485
4D SQL Server One Conn. Licence	Long Integer	825242165
4D View License	Long Integer	808465207
4D Web License	Long Integer	808464945
4D Web Local License	Long Integer	808530481
4D Web One Connection License	Long Integer	825242161
4D Write License	Long Integer	808464697

ISO Latin Character Entities

Constant	Type	Value
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 Latin Character Entities (continued)

Constant	Type	Value
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;
ISO L1 n tilde	String	&ntilde;
ISO L1 o acute	String	&oacute;
ISO L1 o circumflex	String	&ocirc;
ISO L1 o grave	String	&ograve;
ISO L1 o slash	String	&oslash;
ISO L1 o tilde	String	&otilde;
ISO L1 o umlaut	String	&ouml;
ISO L1 Quotation mark	String	&quot;
ISO L1 Registered	String	&reg;
ISO L1 sharp s German	String	&szlig;
ISO L1 thorn Icelandic	String	&thorn;
ISO L1 u acute	String	&uacute;
ISO L1 u circumflex	String	&ucirc;
ISO L1 u grave	String	&ugrave;
ISO L1 u umlaut	String	&uuml;
ISO L1 y acute	String	&yacute;
ISO L1 y umlaut	String	&yuml;

List box

Related command(s): Get listbox information, SELECT LISTBOX ROW.

Constant	Type	Value
Add to listbox selection	Long Integer	1
Display listbox header	Long Integer	0
Display listbox hor scrollbar	Long Integer	2
Display listbox ver scrollbar	Long Integer	4
Listbox header height	Long Integer	1
Listbox hor scrollbar height	Long Integer	3
Listbox ver scrollbar width	Long Integer	5
Position listbox hor scrollbar	Long Integer	6
Position listbox ver scrollbar	Long Integer	7
Remove from listbox selection	Long Integer	2
Replace listbox selection	Long Integer	0

Math

Related command(s): Arctan, Cos, Sin, Tan.

Constant	Type	Value
Degree	Real	0.0174532925199432958
e number	Real	2.71828182845904524
Pi	Real	3.141592653589793239
Radian	Real	57.2957795130823088

Menu item properties

Constant	Type	Value
Access Privileges	String	4D_access_group
Associated Standard Action	String	4D_standard_action
Start a New Process	String	4D_start_new_process

Object alignment

Related command(s): Get alignment, SET ALIGNMENT.

Constant	Type	Value
Align default	Long Integer	1
Align left	Long Integer	2
Align right	Long Integer	4
Center	Long Integer	3

Open form window

Related command(s): Open form window.

Constant	Type	Value
At the Bottom	Long Integer	393216
At the Top	Long Integer	327680
Horizontally Centered	Long Integer	65536
Modal dialog box	Long Integer	1
Movable dialog box	Long Integer	5
On the Left	Long Integer	131072
On the Right	Long Integer	196608
Palette window	Long Integer	1984
Plain window	Long Integer	8
Pop up form window	Long Integer	32
Sheet form window	Long Integer	33
Vertically Centered	Long Integer	262144

Open window

Related command(s): Open external window, Open window.

Constant	Type	Value
Alternate dialog box	Long Integer	3
Has grow box	Long Integer	4
Has highlight	Long Integer	1
Has window title	Long Integer	2
Has zoom box	Long Integer	8
Metal Look	Long Integer	2048
Modal dialog box	Long Integer	1
Movable dialog box	Long Integer	5
Palette window	Long Integer	1984
Plain dialog box	Long Integer	2
Plain fixed size window	Long Integer	4
Plain no zoom box window	Long Integer	0
Plain window	Long Integer	8
Pop up window	Long Integer	32
Resizeable sheet window	Long Integer	34
Round corner window	Long Integer	16
Sheet window	Long Integer	33

Pasteboard

Related command(s): APPEND DATA TO PASTEBOARD, GET PASTEBOARD DATA, Pasteboard data size.

Constant	Type	Value
No such data in pasteboard	Long Integer	-102
Picture data	String	PICT
Text data	String	TEXT

Picture Compression

Related command(s): COMPRESS PICTURE, COMPRESS PICTURE FILE, LOAD COMPRESS PICTURE FROM FILE.

Constant	Type	Value
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

Related command(s): CREATE THUMBNAIL, Get format, SET FORMAT.

Constant	Type	Value
On Background	Long Integer	3
Replicated	Long Integer	7
Scaled to Fit	Long Integer	2
Scaled to fit prop centered	Long Integer	6
Scaled to fit proportional	Long Integer	5
Truncated Centered	Long Integer	1
Truncated non Centered	Long Integer	4

Picture Transformation

Constant	Type	Value
Crop	Long Integer	100
Fade to grey scale	Long Integer	101
Flip horizontally	Long Integer	3
Flip vertically	Long Integer	4
Horizontal concatenation	Long Integer	1
Reset	Long Integer	0
Scale	Long Integer	1
Superimposition	Long Integer	3
Translate	Long Integer	2
Vertical concatenation	Long Integer	2

Platform Interface

Related command(s): Get platform interface, SET PLATFORM INTERFACE.

Constant	Type	Value
Automatic Platform	Long Integer	-1
Mac OS 7	Long Integer	0
Mac OS 9	Long Integer	3
Mac Theme	Long Integer	4
Windows 3.11, NT 3.51	Long Integer	1
Windows 9x	Long Integer	2

Platform Properties

Related command(s): PLATFORM PROPERTIES.

Constant	Type	Value
Intel Compatible	Long Integer	586
Mac OS	Long Integer	2
Power PC	Long Integer	406
Windows	Long Integer	3
_O_INTEL 386	Long Integer	386
_O_INTEL 486	Long Integer	486
_O_Macintosh 68K	Long Integer	1
_O_PowerPC 601	Long Integer	601
_O_PowerPC 603	Long Integer	603
_O_PowerPC 604	Long Integer	604
_O_PowerPC G3	Long Integer	510

Print options

Related command(s): GET PRINT OPTION, PRINT OPTION VALUES, SET PRINT OPTION.

Constant	Type	Value
Color option	Long Integer	8
Destination option	Long Integer	9
Double sided option	Long Integer	11
Hide printing progress option	Long Integer	14
Mac spool file format option	Long Integer	13
Number of copies option	Long Integer	4
Orientation option	Long Integer	2
Paper option	Long Integer	1
Paper source option	Long Integer	5
Scale option	Long Integer	3
Spooler document name option	Long Integer	12

Process state

Related command(s): PROCESS PROPERTIES, Process state.

Constant	Type	Value
Aborted	Long Integer	-1
Delayed	Long Integer	1
Does not exist	Long Integer	-100
Executing	Long Integer	0
Hidden modal dialog	Long Integer	6
Paused	Long Integer	5
Waiting for input output	Long Integer	3
Waiting for internal flag	Long Integer	4
Waiting for user event	Long Integer	2

Process Type

Related command(s): PROCESS PROPERTIES.

Constant	Type	Value
Apple Event Manager	Long Integer	-7
Cache Manager	Long Integer	-4
Created from Menu Command	Long Integer	2
Created from Programming	Long Integer	1
Created from User Mode	Long Integer	3
Design Process	Long Integer	-2
Event Manager	Long Integer	-8
External Task	Long Integer	-9
Indexing Process	Long Integer	-5
None	Long Integer	0
Other 4D Process	Long Integer	-10
Other User Process	Long Integer	4
Serial Port Manager	Long Integer	-6
User or Custom Menus Process	Long Integer	-1
Web Process with Context	Long Integer	-11
Web Process with no Context	Long Integer	-3

QR Area Properties

Related command(s): QR Get area property, QR SET AREA PROPERTY.

Constant	Type	Value
qr view color toolbar	Long Integer	5
qr view column toolbar	Long Integer	6
qr view contextual menus	Long Integer	7
qr view menubar	Long Integer	1
qr view operators toolbar	Long Integer	4
qr view standard toolbar	Long Integer	2
qr view style toolbar	Long Integer	3

QR Borders

Related command(s): QR GET BORDERS.

Constant	Type	Value
qr bottom border	Long Integer	8
qr inside horizontal border	Long Integer	32
qr inside vertical border	Long Integer	16
qr left border	Long Integer	1
qr right border	Long Integer	4
qr top border	Long Integer	2

QR Commands

Related command(s): QR EXECUTE COMMAND, QR Get command status.

Constant	Type	Value
qr cmd 4D View destination	Long Integer	2503
qr cmd add column	Long Integer	2608
qr cmd alt back color palette	Long Integer	1004
qr cmd automatic width	Long Integer	2605
qr cmd average	Long Integer	507
qr cmd back color palette	Long Integer	1003
qr cmd back colors toolbar	Long Integer	2052
qr cmd bold	Long Integer	500
qr cmd borders	Long Integer	2609
qr cmd center justified	Long Integer	504
qr cmd columns toolbar	Long Integer	2054
qr cmd count	Long Integer	510
qr cmd default justified	Long Integer	512
qr cmd delete column	Long Integer	2601
qr cmd disk file destination	Long Integer	2501
qr cmd edit column	Long Integer	2603
qr cmd font color palette	Long Integer	1002
qr cmd font dropdown	Long Integer	1000
qr cmd format	Long Integer	2606
qr cmd generate	Long Integer	2008
qr cmd graph destination	Long Integer	2502
qr cmd header and footer	Long Integer	2005
qr cmd hide column	Long Integer	2602
qr cmd hide line	Long Integer	2607
qr cmd HTML file destination	Long Integer	2504
qr cmd insert column	Long Integer	2600
qr cmd italic	Long Integer	501
qr cmd left justified	Long Integer	503
qr cmd max	Long Integer	509
qr cmd min	Long Integer	508
qr cmd move left	Long Integer	3002
qr cmd move right	Long Integer	3003
qr cmd new	Long Integer	2000
qr cmd open	Long Integer	2001
qr cmd operators toolbar	Long Integer	2051
qr cmd page setup	Long Integer	2006

QR Commands (continued)

Constant	Type	Value
qr cmd plain	Long Integer	511
qr cmd presentation	Long Integer	2611
qr cmd print preview	Long Integer	2007
qr cmd printer destination	Long Integer	2500
qr cmd repeated values	Long Integer	2604
qr cmd revert to save	Long Integer	2004
qr cmd right justified	Long Integer	505
qr cmd save	Long Integer	2002
qr cmd save as	Long Integer	2003
qr cmd standard deviation	Long Integer	513
qr cmd standard toolbar	Long Integer	2053
qr cmd style toolbar	Long Integer	2050
qr cmd sum	Long Integer	506
qr cmd totals spacing	Long Integer	2610
qr cmd underline	Long Integer	502

QR Document Properties

Related command(s): QR Get document property, QR SET DOCUMENT PROPERTY.

Constant	Type	Value
qr printing dialog	Long Integer	1
qr unit	Long Integer	2

QR Operators

Related command(s): QR GET TOTALS DATA, QR SET TOTALS DATA.

Constant	Type	Value
qr average	Long Integer	2
qr count	Long Integer	16
qr max	Long Integer	8
qr min	Long Integer	4
qr standard deviation	Long Integer	32
qr sum	Long Integer	1

QR Output Destination

Related command(s): QR GET DESTINATION, QR SET DESTINATION.

Constant	Type	Value
qr 4D Chart area	Long Integer	4
qr 4D View area	Long Integer	3
qr HTML file	Long Integer	5
qr printer	Long Integer	1
qr text file	Long Integer	2

QR Report Types

Related command(s): QR Get report kind, QR SET REPORT KIND.

Constant	Type	Value
qr cross report	Long Integer	2
qr list report	Long Integer	1

QR Rows for Properties

Related command(s): QR GET BORDERS, QR Get info row, QR SET BORDERS, QR SET INFO ROW, QR SET TEXT PROPERTY.

Constant	Type	Value
qr detail	Long Integer	-2
qr footer	Long Integer	-5
qr grand total	Long Integer	-3
qr header	Long Integer	-4
qr title	Long Integer	-1

QR Text Properties

Related command(s): QR Get text property, QR SET TEXT PROPERTY.

Constant	Type	Value
qr alternate background color	Long Integer	9
qr background color	Long Integer	8
qr bold	Long Integer	3
qr font	Long Integer	1
qr font size	Long Integer	2
qr italic	Long Integer	4
qr justification	Long Integer	7
qr text color	Long Integer	6
qr underline	Long Integer	5

Queries

Related command(s): SET QUERY DESTINATION.

Constant	Type	Value
Description in Text Format	Long Integer	0
Description in XML Format	Long Integer	1
Into current selection	Long Integer	0
Into named selection	Long Integer	2
Into set	Long Integer	1
Into variable	Long Integer	3

Relations

Related command(s): GET FIELD RELATION, SET FIELD RELATION.

Constant	Type	Value
Automatic	Long Integer	3
Do not modify	Long Integer	0
Manual	Long Integer	2
No relation	Long Integer	0
Structure configuration	Long Integer	1

Resources Properties

Related command(s): Get resource properties, SET RESOURCE PROPERTIES.

Constant	Type	Value
Changed resource bit	Long Integer	1
Changed resource mask	Long Integer	2
Locked resource bit	Long Integer	4
Locked resource mask	Long Integer	16
Preloaded resource bit	Long Integer	2
Preloaded resource mask	Long Integer	4
Protected resource bit	Long Integer	3
Protected resource mask	Long Integer	8
Purgeable resource bit	Long Integer	5
Purgeable resource mask	Long Integer	32
System heap resource bit	Long Integer	6
System heap resource mask	Long Integer	64

SCREEN DEPTH

Related command(s): SCREEN DEPTH, SET SCREEN DEPTH.

Constant	Type	Value
Black and white	Long Integer	0
Four colors	Long Integer	2
Is color	Long Integer	1
Is gray scale	Long Integer	0
Millions of colors 24 bit	Long Integer	24
Millions of colors 32 bit	Long Integer	32
Sixteen colors	Long Integer	4
Thousands of colors	Long Integer	16
Two fifty six colors	Long Integer	8

SET RGB COLORS

Related command(s): SET RGB COLORS.

Constant	Type	Value
Default background color	Long Integer	-2
Default dark shadow color	Long Integer	-3
Default foreground color	Long Integer	-1
Default light shadow color	Long Integer	-4
Disable highlight item color	Long Integer	-11
Highlight menu background color	Long Integer	-9
Highlight menu text color	Long Integer	-10
Highlight text background color	Long Integer	-7
Highlight text color	Long Integer	-8

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.

Related command(s): APPEND DATA TO PASTEBOARD, GET PASTEBOARD DATA, Get resource properties, Pasteboard data size, SET RESOURCE PROPERTIES.

Constant	Type	Value
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

Related command(s): Open document, Select document, Test path name.

Constant	Type	Value
Alias selection	Long Integer	8
Get Pathname	Long Integer	3
Is a directory	Long Integer	0
Is a document	Long Integer	1
Multiple files	Long Integer	1
Package open	Long Integer	2
Package selection	Long Integer	4
Read and Write	Long Integer	0
Read Mode	Long Integer	2
Use sheet window	Long Integer	16
Write Mode	Long Integer	1

System Folder

The folders corresponding to the constants 4 to 11 no longer exist under Mac OS X (they were only used under Mac OS 9). When these constants are used under Mac OS, System Folder returns an empty string.
Constants 6, 7, 10 and 11 (Mac OS only) are therefore completely obsolete starting with version 2004 of 4D. Constants 4, 5, 8 and 9 can nevertheless still be used under Windows.

Related command(s): System folder.

Constant	Type	Value
Apple or Start Menu_All	Long Integer	8
Apple or Start Menu_User	Long Integer	9
Desktop Win	Long Integer	15
Favorites Win	Long Integer	14
Fonts	Long Integer	1
Mac Control Panels	Long Integer	11
Mac Extensions	Long Integer	10
Mac Shutdown Items_All	Long Integer	6
Mac Shutdown Items_User	Long Integer	7
Preferences or Profiles_All	Long Integer	2
Preferences or Profiles_User	Long Integer	3
Program Files Win	Long Integer	16
Startup Items_All	Long Integer	4
Startup Items_User	Long Integer	5
System	Long Integer	0
System Win	Long Integer	12
System32 Win	Long Integer	13

System format

Constant	Type	Value
Currency symbol	Long Integer	2
Date separator	Long Integer	13
Decimal separator	Long Integer	0
Short date day position	Long Integer	15
Short date month position	Long Integer	16
Short date year position	Long Integer	17
System date long pattern	Long Integer	8
System date medium pattern	Long Integer	7
System date short pattern	Long Integer	6
System time AM label	Long Integer	18
System time long pattern	Long Integer	5
System time medium pattern	Long Integer	4
System time PM label	Long Integer	19
System time short pattern	Long Integer	3
Thousand separator	Long Integer	1
Time separator	Long Integer	14

TCP Port Numbers

Related command(s): Get database parameter, SET DATABASE PARAMETER.

Constant	Type	Value
TCP Authentication	Long Integer	113
TCP DNS	Long Integer	53
TCP Finger	Long Integer	79
TCP FTP Control	Long Integer	21
TCP FTP Data	Long Integer	20
TCP Gopher	Long Integer	70
TCP HTTP WWW	Long Integer	80
TCP IMAP3	Long Integer	220
TCP Kerberos	Long Integer	88
TCP KLogin	Long Integer	543
TCP Nickname	Long Integer	43
TCP NNTP	Long Integer	119
TCP NTalk	Long Integer	518
TCP NTP	Long Integer	123
TCP PMCP	Long Integer	1643
TCP PMD	Long Integer	1642
TCP POP3	Long Integer	110
TCP Printer	Long Integer	515
TCP RADACCT	Long Integer	1646
TCP RADIUS	Long Integer	1645
TCP Remote Cmd	Long Integer	514
TCP Remote Exec	Long Integer	512
TCP Remote Login	Long Integer	513
TCP Router	Long Integer	520
TCP SMTP	Long Integer	25
TCP SNMP	Long Integer	161
TCP SNMPTRAP	Long Integer	162
TCP SUN RPC	Long Integer	111
TCP Talk	Long Integer	517
TCP Telnet	Long Integer	23
TCP TFTP	Long Integer	69
TCP UUCP	Long Integer	540
TCP UUCP RLOGIN	Long Integer	541

Time Display Formats

Related command(s): Get format, SET FORMAT, String.

Constant	Type	Value
Blank if null time	Long Integer	100
HH MM	Long Integer	2
HH MM AM PM	Long Integer	5
HH MM SS	Long Integer	1
Hour Min	Long Integer	4
Hour Min Sec	Long Integer	3
ISO Time	Long Integer	8
Min Sec	Long Integer	7
MM SS	Long Integer	6
System time long	Long Integer	11
System time long abbreviated	Long Integer	10
System time short	Long Integer	9

Value for Associated Standard Action

Related command(s): GET MENU ITEM PROPERTY, SET MENU ITEM PROPERTY.

Constant	Type	Value
Accept Action	Long Integer	2
Add subrecord Action	Long Integer	14
Cancel Action	Long Integer	1
Clear Action	Long Integer	21
Copy Action	Long Integer	19
Cut Action	Long Integer	18
Delete record Action	Long Integer	7
Delete subrecord Action	Long Integer	13
Edit subrecord Action	Long Integer	12
First page Action	Long Integer	10
First record Action	Long Integer	5
Last page Action	Long Integer	11
Last record Action	Long Integer	6
MSC Action	Long Integer	36
Next page Action	Long Integer	8
Next record Action	Long Integer	3
No Action	Long Integer	0
Paste Action	Long Integer	20
Preferences Action	Long Integer	32
Previous page action	Long Integer	9
Previous record Action	Long Integer	4
Quit Action	Long Integer	27
Redo Action	Long Integer	31
Return to Design mode	Long Integer	35
Select all Action	Long Integer	22
Show Clipboard Action	Long Integer	23
Test Application Action	Long Integer	26
Undo Action	Long Integer	17

Web Services (Client)

Related command(s): CALL WEB SERVICE, Get Web Service error info, SET WEB SERVICE OPTION.

Constant	Type	Value
Web Service Detailed Message	Long Integer	1
Web Service Dynamic	Long Integer	0
Web Service Error Code	Long Integer	0
Web Service Fault Actor	Long Integer	3
Web Service HTTP Error code	Long Integer	2
Web Service HTTP Timeout	Long Integer	1
Web Service Manual	Long Integer	3
Web Service Manual In	Long Integer	1
Web Service Manual Out	Long Integer	2
Web Service SOAP Header	Long Integer	2
Web Service SOAP Version	Long Integer	3
Web Service SOAP_1_1	Long Integer	0
Web Service SOAP_1_2	Long Integer	1

Web Services (Server)

Related command(s): Get SOAP info, SEND SOAP FAULT, SOAP DECLARATION.

Constant	Type	Value
SOAP Client Fault	Long Integer	1
SOAP Input	Long Integer	1
SOAP Method Name	Long Integer	1
SOAP Output	Long Integer	2
SOAP Server Fault	Long Integer	2
SOAP Service Name	Long Integer	2

Window kind

Related command(s): Window kind.

Constant	Type	Value
External window	Long Integer	5
Floating window	Long Integer	14
Modal dialog	Long Integer	9
Regular window	Long Integer	8

Windows Log Events

Related command(s): LOG EVENT.

Constant	Type	Value
Error Message	Long Integer	2
Information Message	Long Integer	0
Warning Message	Long Integer	1

XML

Related command(s): DOM Get XML information, SAX Get XML node.

Constant	Type	Value
DOCTYPE Name	Long Integer	3
Document URI	Long Integer	6
Encoding	Long Integer	4
PUBLIC ID	Long Integer	1
SYSTEM ID	Long Integer	2
Version	Long Integer	5
XML CDATA	Long Integer	7
XML Comment	Long Integer	2
XML DATA	Long Integer	6
XML End Document	Long Integer	9
XML End Element	Long Integer	5
XML Entity	Long Integer	8
XML Processing Instruction	Long Integer	3
XML Start Document	Long Integer	1
XML Start Element	Long Integer	4

Command Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y