# 4D Plug-in Wizard User Manual

by Christian Cypert and Francois Marchal

## INTRODUCTION

### What is the 4D Plug-In Wizard?

The 4D Plug-In Wizard was created to help in the creation of 4th Dimension plug-ins. The API is supported by the 4D Plug-In Wizard. The Wizard helps in creating a basic framework for the API so that a developer does not have to worry about how to set up the Project, header files, and such. This is all done automatically for the developer.

The plug-in developer creates a project for the plug-in, then enters the desired names, parameters, and return values of the routines. The developer arranges the routines by theme, or can tell the Wizard to prepare the creation of an external area. Once this is done, the 4D Plug-In Wizard generates all the necessary files: source files, resource files, export files, and project files (for both MacOS and Windows if necessary). The source code of the plug-in is generated, containing all necessary prototypes, constants, parameters, and code skeletons. The 4D Plug-In Wizard is very useful, particularly for Windows developers, since it generates the appropriate resources ('4BNX', 'STR#'). Once the 4D Plug-In Wizard has generated the files, the developer can open the project and start to write the plug-in code. Each time the developer needs to use a 4D Plug-In API function, the developer calls the appropriate routine, (stored in 4DPlug-inAPI) and is ready for compilation. If the plug-in developer needs to change the syntax of a routine or add routines, etc., the developer can use the 4D Plug-In Wizard to generate the new resource file. The developer can also cut and paste parts of the newly generated source code into the source code under development.

### Why use the 4D Plug-In Wizard?

Those who are already familiar with creating and writing plug-ins for 4th Dimension may not wish to use the Wizard. For the vast majority of developers, this may be their first attempt to create a plug-in. This is where the 4D Plug-In Wizard comes in. By using the 4D Plug-In Wizard the developer does not have to worry about the small things that can cause a Project not to compile. Its always the small things that tend to cause the most frustration when writing code. By using the 4D Plug-In Wizard, we hope to eliminate as many of the small issues as possible that may pop up while writing code. We want to enable developers to spend more time writing code and less time trying to figure out why their projects won't compile because of possible missing headers.

### What is possible and not possible with the 4D Plug-In Wizard?

Do not think that the 4D Plug-In Wizard will write the entire plug-in. This is not what it was designed to accomplish. It is up to the developer to write in the missing pieces. For example, the 4D Plug-In Wizard will create a .c and .h file that will contain the Themes, which will be explained later, and Commands for a plug-in. It is up to the developer to actually write the code for the commands. The 4D Plug-In Wizard sets up all the calls to the commands, and the return and input parameters as well, but it will be up to the developer to write the core of the code. Once a developer becomes familiar with how plug-ins are created, he or she may no longer wish to use the 4D Plug-In Wizard. This is fine. The developer does not need to use the 4D Plug-In Wizard to create a plug-in. We recommend using the Wizard since it formalizes the structure of the plug-in to a larger extent, and will automatically write several pieces of code that developers would have to write anyway.

### Description of various files used by the 4D Plug-In Wizard

**4D Plugin API {folder}:** This folder contains the various 4D API .c, .h, and .def files needed to compile a 4D plug-in. These files consist of four headers, a source file, and a definition file needed for Windows.

**4D PlugIn Wizard and 4D Plug-In Wizard.data:** This is the 4D Plug-In Wizard application.

**4D Runtime v6.7:** A copy of 4D Runtime is provided in case the developer does not have access to 4D.

**Mac4DX {folder}:** This folder contains a file that supports the 4D Plug-In Wizard. This folder must not be removed, otherwise the 4D Plug-In Wizard will not function properly.

**Win4DX {folder}:** This folder contains the .RSR file for a 4D plug-in for Windows. Since the resource and data fork are separate under Windows, we have provided the .RSR file for the developer.

**Online Documentation.html:** This file loads the 4D Web site on APIs. If developers have any questions regarding a particular API call, they should use this .html file to look up the command in question.

## DESCRIPTION OF THE FRAMEWORK OF A 4D PLUG-IN

### Parameters and Returned Values

The main routine of the plug-in must be called PluginMain. When 4D calls the plug-in, it calls a routine named FourDPack under Windows and Main on Macintosh. FourDPack and Main are implemented in the 4DPluginAPI.c source file, and they call the Plugin Main routine provided by the plug-in.

#### Parameters

A routine can receive up to 25 parameters. The parameters are received (among other things) in a PA_PluginBlock structure. A pointer to this structure (PA_PluginParameters) is passed to PluginMain. The 4D Plug-in API gives a developer utility functions to access the parameters, making it easy to get or set the value of a parameter. Simply call the appropriate API function and then give it the number of the parameter to access. As parameters are passed by reference, it is also possible to change the value of a parameter. The syntax of those API functions are GetXXXParameter and SetXXXParameter, where XXX is the type of the parameter. Example: PA_GetLongParameter, PA_SetLongParameter, PA_GetStringParameter.

As an example, if a routine wants to read the value of parameter 3 that is a long, it can use:

```
aLong = PA_GetLongParameter(params, 3);
```

If a routine wants to set the value of parameter 2 that is a short, it can use:

```
PA_SetShortParameter(params, 2, aShort);
```

#### Returned Values

Similar to parameters, the API gives a developer routines (PA_ReturnXXX) to return values of any kind. This must be used when the routine is declared as returning a value (in 4D Plug-In Wizard). If a routine returns a string, it needs to use PA_ReturnString(parameters,theString).

#### Skeleton of a plug-in source code

The Plugin Main routine switches the selector value and calls a routine for each case, passing the PA_PluginParameters received to the routine. Each routine then deals with the parameters and the returned value.

Let's look at sample code of this dispatch as it can be generated by the 4D Plug-In Wizard.

```
Void PluginMain( long selector, PA_PluginParameters params )
{
switch( selector )
  {
case 1:
aRoutine( params );
break;

case 2:
anotherRoutine( params );
break;
  }
}

void aRoutine( PA_PluginParameters params )
{
// Code of first routine. It expects a long and a string
  long   aLong;
  char   aString[256];

// We get the parameters
aLong = PA_GetLongParameter( params, 1 );
PA_GetStringParameter( params, 2, aString );

// . . . the code . . .
}

void anotherRoutine( PA_PluginParameters params )
{
// Say this one returns a numeric value
double   theDouble;
// . . . the code . . .
PA_ReturnDouble ( params, theDouble );
}
```

## HOW TO USE THE 4D PLUG-IN WIZARD

The 4D Plug-In Wizard is a straightforward 4D application. For anyone who does not own or use 4D, we have provided 4D Runtime to allow any developer to run the 4D Plug-In Wizard.

To run the 4D Plug-In Wizard, drag the 4D Plug-In Wizard icon over the 4D Runtime icon. When the 4D Plug-In Wizard first comes up, the developer will see a dialog called Projects (see fig.1 on next page). This dialog allows the developer to create a new Plug-in Project, or edit or delete a previous Project. Below is an image and description of this dialog. Notice that any previously created Projects are listed on the left-hand. The New Button creates a new Project. The Edit Button opens a previously created Plug-in Project. The Delete Button deletes the Plug-in Project currently highlighted on the left hand side of the dialog.


*figure 1.*

## New/Edit Project for 4D

When creating a new Project, a dialog will be presented asking what the name of the new Project will be. This name will show up in the Project listing dialog (see fig. 2) when the 4D Plug-In Wizard is opened. After this dialog, the main development dialog is displayed. This is the same dialog that is displayed when editing a previous Project. This development window contains a great deal of information, but is displayed in a straightforward manner. An image of the dialog is shown below.

On the left hand side of this dialog starting from the top down is the name of the Project. The Hiearchical List below displays the Themes and Commands for the Plug-in. Below further are six buttons: Add Theme, Add Command, Add External Area, Delete, Edit Constants, and Import from Plug-in.

**Add Theme:** This button will add a new Theme to the list above. A Theme in 4D is what is displayed in the lower right hand corner of the 4D Method Editor. When clicking on a Theme, the various commands are then displayed.

**Add Command:** This button adds a new Command based on the currently highlighted Theme.

**Add External Area:** This button adds an External Area to the 4D plug-in. This will be explained in more detail later.

**Delete:** This button deletes Themes, Commands, and External Areas. If a Theme is deleted, the associated Commands are also deleted.

**Edit Constants:** This button allows the developer to edit or add constants to the plug-in. This will be explained in more detail later.

**Import from Plug-in:** This button allows the developer to import constants in from another plug-in to be used by the current plug-in. In the middle of the dialog (fig. 2) there is a table dialog that displays the text of the 4DPlug-in.c and 4DPlugin.h. When adding a Theme, Command, or External Area, notice that the 4D Plug-In Wizard is creating the framework code automatically. Below this table area are several options for the developer.

**Preview Area Live Update:** This checkbox in the upper right hand corner will turn off the automatic updating of the 4DPlugin.c and 4DPlugin.h areas.

**Use Ansi:** This checkbox will add the header files "stdio.h" and "string.h" to the 4DPlugin.c file.

**Use Win32 API:** This checkbox will add the header file "Windows.h" to the 4Dplugin.c file.

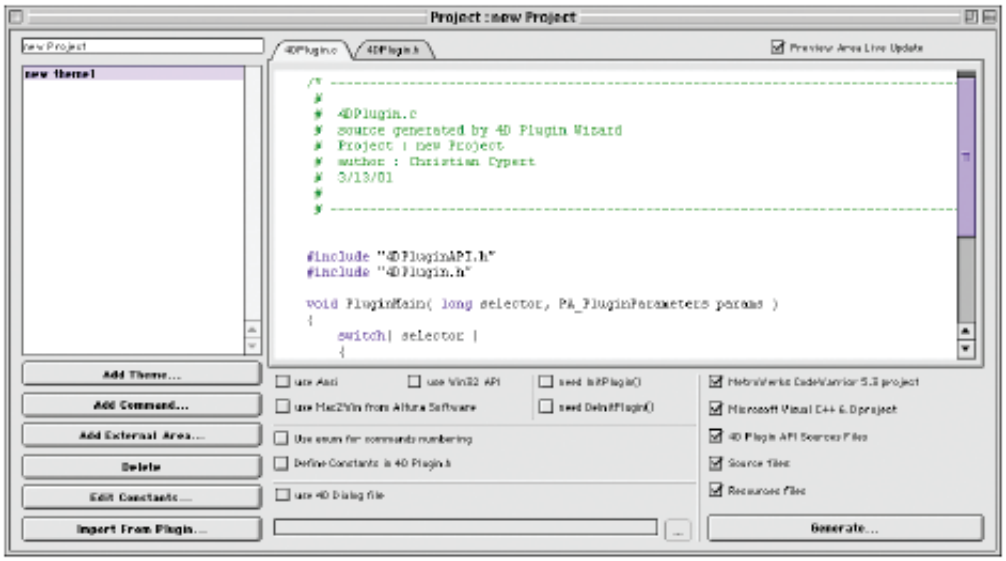


*figure 2.*

**Use Mac2Win from Altura Software:** This checkbox will add the header file "AsiPort.h" to the 4Dplugin.c file. This allows for the use of Altura calls from Windows.

**Need InitPlugin():** This checkbox will add the InitPlugin phase to the code automatically. This is needed if the plug-in needs to run code when initialized.

**Need DeInitPlugin():** This checkbox will add the DeInitPlugin phase to the code automatically. This is needed if the plug-in needs to run code when it is being de-initialized.

**Use enums for command numbering:** The checkbox changes the numbering of the main switch statement from numbers to enums.

**Define constants in 4D Plugin.h:** If this checkbox is marked when the plug-in is created, adding constants to the plug-in will cause the constants to be defined in the 4Dplugin.h file.

**Use 4D Dialog File:** This checkbox will add a 4D Dialog (resource 'FO4D') to the plug-in. This option is needed if a developer wishes to use any of the "Form as a Dialog" API calls. These API calls allow the developer to display a 4D Dialog that has been saved as a FO4D resource. To specify the file, one would need to click on the button next to the text area (button with elipses).

**Metrowerks CodeWarrior 5.3 Project:** Use this option to create a CodeWarrior v5.3 Project based on the Plug-in Code.

**Microsoft Visual C++ 6.0 Project:** Use this option to create a Visual C++ v6.0 project based on the Plug-in Code.

**4D Plug-in API Source Files:** This option will copy the 4D API source files into the newly created 4D Plug-in Project folder.

**Source Files:** This checkbox will produce the source files (.c, .h) needed. If not checked when the code is generated, the source code files will not be produced.

**Resource Files:** This checkbox will produce the resource files needed. If not checked when the code is generated, the resource files will not be produced.

**Generate Button:** This button creates a folder on the hard drive that contains all the source files and projects that are needed to work with and compile the new 4D Plug-in.

## Adding Themes and Commands

The first thing a developer must decide is how each plug-in call will be configured. How many parameters will be passed into and out of the plug-in? Will the plug-in return a value? These questions should be answered before adding the commands for the plug-in. Once the different commands have been decided upon, you are ready to proceed. First, add a Theme to the plug-in. This is accomplished by clicking on the Add Theme button. A dialog box will be displayed asking what the Theme's name will be. This is the name that will appear in the lower right hand corner of the method editor in 4D. When clicking on a Theme, a popup will appear displaying all the commands associated with the theme. Once a Theme has been created, you can now add a Command. **Note:** Since many Themes can be added to the plug-in, make sure the correct one is highlighted before clicking on the Add Command button.

**Steps to adding a command:**

1. Click on the Add Command button. A dialog (fig 3.) is displayed.

   Within this dialog, the parameters for the plug-in are created. The name of the command is specified at the top of the dialog. Just to the right of the Command Name is the return value, if one exists. If the return value is not specified, then the plug-in command will not return a value. This would be similar to the 4D command NEW PROCESS. The NEW PROCESS command returns the number of the newly created process.

2. (Optional) Click on the return type to specify what type of value is to be returned from the new command.

   Within the section called Parameters is where the parameters to a plug-in command are specified. The Command may not need any parameters. This would be similar to the BEEP command.

3. Click on the Add button. A parameter will then be included in the parameters list.

   The Remove Button deletes the parameter currently highlighted. When a parameter has been added, there are three areas that can be modified.

**Modifiable Areas of a Parameter**

Type of parameter: By clicking on the type pop-up, a developer can specify if the parameter is to be a string, text, integer, real, and so on.

Direction of parameter: A developer can set the direction of the parameter. He or she can decide whether the parameter will send information into the plug-in, whether the it will simply be used to return information, or whether it will do both.

Parameter name: A developer can modify the name of the parameter by double clicking on the name.

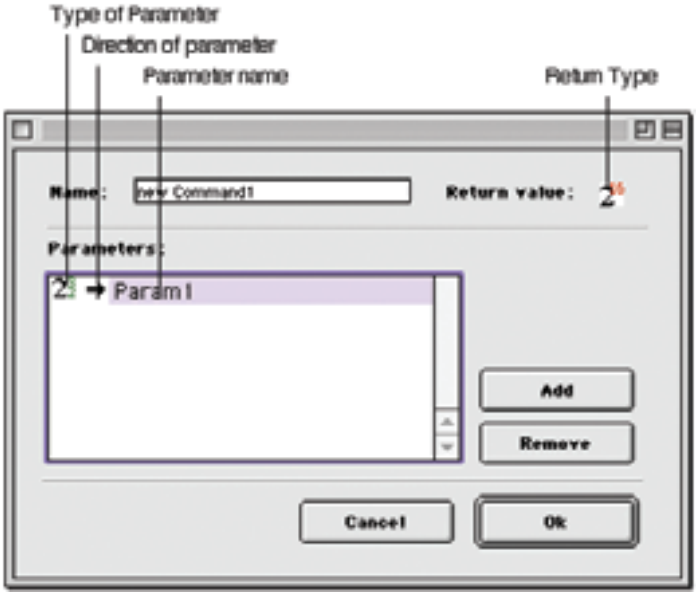


*figure 3.*

4. When finished with the different parameters, simply click on the OK button and the command with the specified parameters will be added to the source code.

## Adding External Areas

To add an external area, click on the "Add External Area..." button. Clicking on this button will present the following dialog (fig 4):

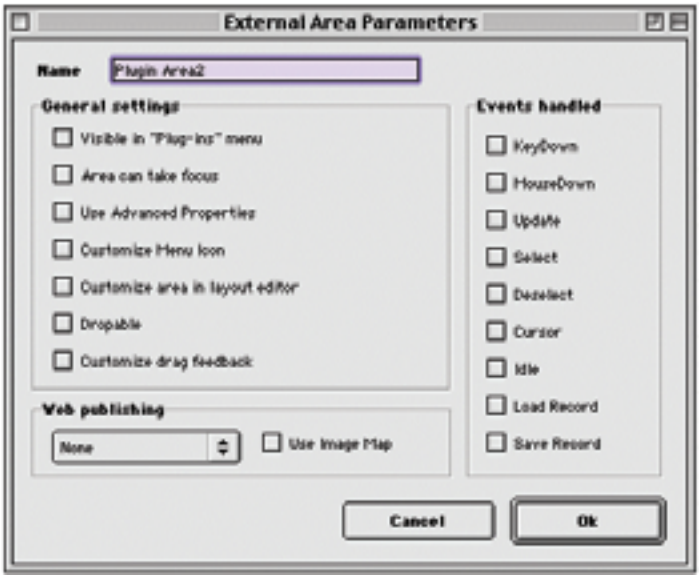


*figure 4.*

**Name:** Name given to the plug-in as it will appear on the layout editor, and "Plug-in" menu

**General Settings**

Visible in "Plug-in" menu: Will display the "Name" of the plug-in in the "Plug-in" menu.

Area can take focus: The Plug-in area is focusable.

Use Advanced Properties: The area can display the Advanced Properties dialog.

Customize Menu Icon: A developer can set the icon that will appear beside the area name in the tools palette plug-in area pop-up.

Customize area in layout editor: A developer can draw the plug-in in the form editor. By default, the standard 4D Plug-in image will be used.

Dropable: A developer can drag information into the plug-in area.

Customize drag feedback: A developer can allow the plug-in to handle the drag feedback. By default, 4D will handle the drag feedback.

**Events:** Each checkbox represents a different event that the plug-in will accept. Check the appropriate event boxes that the plug-in should detect.

Keydown: A key is pressed. Area can call PA_CallPluginAreaMethod

Mousedown: The mouse button is down. Area can call PA_CallPluginAreaMethod

Update: Area must be updated, redrawn

Select: The user wants to select the area (PA_AcceptSelect)

Deselect: The user wants to leave the area (PA_AcceptDeselect)

Cursor: Mouse has moved (even if the area is not selected) over the area

Idle: The user does nothing. The plug-in could, as an example, draw a clock. A polite plug-in should not take too much time on this event. Area can call PA_CallPluginAreaMethod

Load Record: A record of the table of the form that owns the area is loaded. The area can get a field of this record (i.e., a field whose name is areaName_, like 4DWrite)

Save Record: The record of the table that owns the area is saved. The area can save its content back to a field.

**Web Publishing:** If the plug-in can also be published on the Web, choose which type by choosing from the pop-up menu.

Use Image Map: Display the plug-in as an image instead of HTML on a Web browser.

When finished with setting up the plug-in area, click on the "OK" button.

### Adding Constants

To add or edit a Constant, click on the "Edit Constants..." button. Below is the dialog (fig 5.) that is displayed:

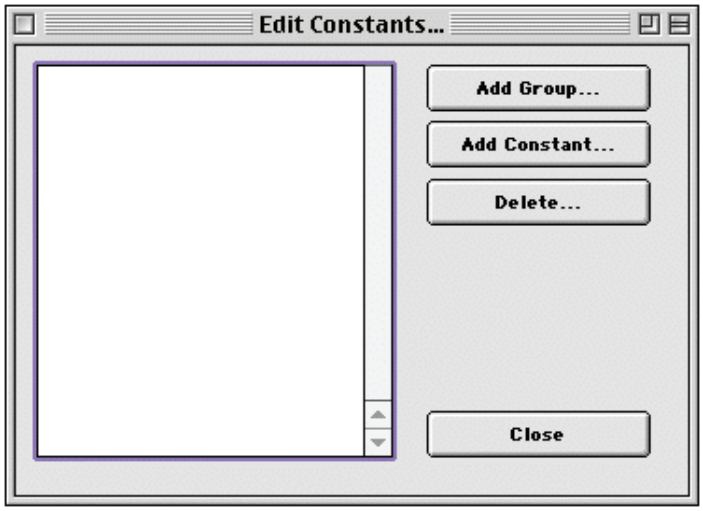


*figure 5.*

**Add Group:** The Add Group button adds a group on the left side which helps categorize the different constants. A developer can create several groups to hold different types of constants.

**Add Constant:** The Add Constant button displays the dialog shown in figure 6. Clicking on this dialog will add a constant to the currently highlighted group.

**Delete:** Deletes a group or constant, depending on which is highlighted.

**Close:** Closes the window.

**Name:** The name of the constant

**Type:** The type of constant (example: string, real, integer, etc.)

**Value:** The value of the constant.

**Define constants in 4Dplugin.h:** This checkbox on the main development screen will define the newly created constants in the Header file. Developers may wish to use this option if they desire.

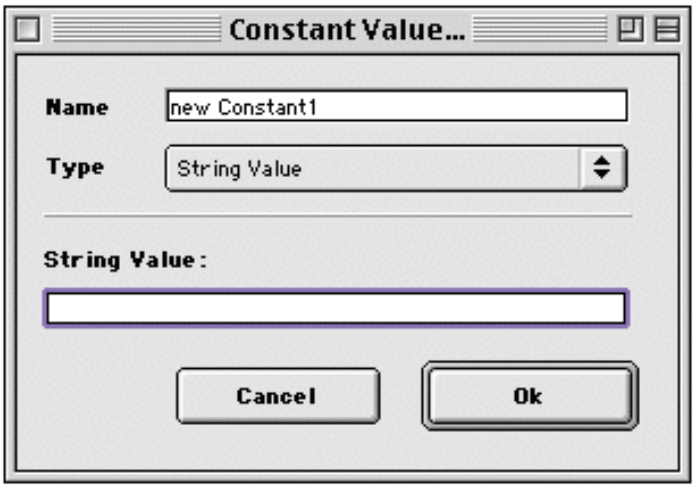


*figure 6.*

### Generating Code to Disk

When the plug-in Themes and Commands have been setup, it is now time to determine which options should be checked when producing the Source Code. As mentioned above there are several checkboxes that can be set. When clicking on the different checkboxes, different types of code and files can be produced.

**Metrowerks Codewarrior 5.3 Project:** Generates a Codewarrior project that is compatible with v5.3 and greater.

**Microsoft Visual C++ 6.0 Project:** Generates a Visual C++ project that is compatible with v6.0 or greater.

**4D Plug-in API Source Files:** Generates the 4D API source files to be used in compiling the plug-in.

**Source Files:** Generates the .c and .h files of the plug-in.

**Resource Files:** Generates the resource files needed to compile the plug-in. These files contain the Theme and Command Syntax information that is displayed in the Method Editor.

Any combination of the checkboxes can be used. For example, if a developer changes only items that would appear in the .c or .h file, then he or she can simply output the Source Files and ignore the rest.

When the different options have been set, click on the Generate button. This button will ask where the files should be saved. Once completed, an alert will be displayed letting a developer know that the files are ready for use. At this point, a developer can open the newly created plug-in project with Codewarrior for the Mac or Visual C++ for Windows, depending on which option was choosen in the 4D Plug-In Wizard.