



>

Jazz Params Plug-in

>

Updated 10 September, 2005

◇

Introduction

◆ Overview

This plugin provides a mechanism whereby parameter information can be passed between FileMaker scripts, irrespective of whether the script lives in the same database or not. Named local variables can also be maintained within the same system.

Due to the limited plug-in framework available, this solution requires that a strict pattern of use be adhered to. With a little practise this overhead becomes second nature. The solution, although not ideal, is far superior to present work-arounds, such as accessing global fields through a relationship, or passing information via the clipboard.

Standard use of the plug-in can be obtained by following the examples and guidelines given in the chapter on *general use*, with only occasional reference to the function summaries when required. This is the recommended method for learning how to use this plugin.

Prior to its release as shareware, this plugin has been used in approximately 20 commercial cross-platform CD-ROMs, and in excess of 100,000 seatings. It has proved to be a very valuable and reliable plugin for these products.

◆ Shareware or Freeware

This plug-in is shareware. It will remind you of this fact occasionally unless you register. (If you make heavy use of this plug-in it will remind you often). Registration does not cost much, and encourages me to write more plugins. If you use this plug-in regularly, please register it by going to <http://www.jazzmedia.com.au>.

The simple license is for use by a single user. If you wish to distribute this plugin with your solution there are separate licenses available on a per-product basis. Consult the web-site for more details.

◆ Disclaimer

The plug-in is offered "as-is", and comes with no implied warranty. You are responsible for determining the plugin's suitability for your particular purpose or purposes, and that it performs accordingly. Please do not register the plugin until you have so-determined the plugin's suitability and effectiveness as per your requirements.

Conventions

◆ Definitions

The terms *function*, *command* and *routine* are used interchangeably in this document, and refer to an *external* function in the plugin. While technically there is a difference between a *function* and a *command*, *FileMaker* makes no such distinction, so the terms can be interchanged without confusion.

The terms *script* and *sub-script* will be used to refer to *Filemaker* scripts, as is consistent with *Filemaker* terminology. Occasionally the documentation may inadvertently refer to a *Filemaker* script as a *function*, *command* or *routine*. If so, context should make it clear whether an *external* (plugin) function or a *sub-script* is being referenced.

◆ Calling functions

This extension makes use of the *FileMaker* plugin architecture. Plugin functions are designed to take exactly one argument (a single string of text) and return a single argument¹. As some of the functions of this plugin require multiple or no arguments, and some of them don't return any value, some encoding of the information needs to take place to conform to the *FileMaker* plugin requirements.

Plugins are designed to be used inside formulas. A simple function to calculate the square of a number could be included inside a calculation like:

```
Set Field ["Area", "3.1415 * External("Square", Radius)"]
```

This script step calculates the radius of a circle, where *Area* and *Radius* are *FileMaker* fields, and *Square* is the name of the function. In this example, the function accepts the field *Radius* as its argument, and returns the square of this number, which is used in the formula as shown.

When an external function does not return a value² (or the value is unimportant) there are two easy ways to use it. The first is to create an *If-End If* pair of commands, with no script steps in-between. Because the body of the *If-End If* statement is empty, the return value of the function is ignored. A *display dialog* function could be called this way:

```
If ["External("Display Dialog", "Hello there)"]  
End If
```

An more compact way to achieve the same result is to create a global field called *temp* (for example) which is reserved for offloading unwanted function results. The same *display dialog* function would now be called like this:

```
Set Field ["temp", "External("Display Dialog", "Hello there)"]
```

¹ Actually plugins accept two arguments (strings), where the first one is the name of the function, and the second one is the data. It is more convenient to consider the first string simply as the name of the function, and thus the second string is the only data passed to this command. When we talk about a single argument being passed to a function, we are referring to this second string.

² Actually all external functions return a value, irrespective of whether one is required. If a return value is not required, by convention the function will return an empty string ("").

This is the method I prefer, because of its compactness, and will be used in the examples throughout this document.

General use

◆ Example 1 — Sending parameters to a sub-script

The following script sends a movie title to a script called *Save Movie(title)*. The script accepts the value, but does nothing with it. You would not typically write such useless scripts. It simply demonstrates the mechanism used to pass values to a sub-script. If you envisage the scripts living in two different databases, this example might come closer to being useful. The example assumes the *FileMaker* field *temp* exists in the database(s) used:

```
Set Field ["temp", "External("Param Send", "Star Wars)"]
PerformScript[Sub-scripts, "Save Movie(title)"]
```

The script sends the value "Star Wars" to the sub-script. The field *temp* in the above script is required only to allow us to call the plug-in, otherwise it is not used in the script. Note that the sub-script has the parameter as part of its name: it is called *Save Movie(title)* instead of simply *Save Movie*. This is a convention only, and recommended as it reminds you what values to supply to the sub-script when calling it. The sub-script *Save Movie(title)* follows:

```
Script Save Movie(title)
Set Field ["temp", "External("Param Begin", "title)"]
Set Field ["temp", "External("Param Get", "title)"]
...
Set Field ["temp", "External("Param End", "")"]
```

The first line indicates to the plugin that a new sub-script is beginning, and lets the plugin perform internal house-keeping tasks. The argument in the first line names the single parameter it receives, as "title". On the second line the parameter is retrieved by name, and the result placed into *temp*. This line simply demonstrates how to fetch the parameter's value. It does nothing useful with it in this example. The final line tells the plugin that the sub-script is about to end, and allows it to perform internal house-keeping tasks.

Every sub-script which receives (or returns) parameters *must* start with *Param Begin* and must call *Param End* before it exits. Normally these will be called at the first and last script steps of every sub-script, as shown in the above example.

◆ Example 2 — Returning values from a sub-script

The following script calls a sub-script to calculate the area of a rectangle, width 5, height 3. It assumes the *FileMaker* field *temp* exists in the database(s) used:

```
Set Field ["temp", "External("Param Send", "5")"]
Set Field ["temp", "External("Param Send", "3")"]
```

```
PerformScript[Sub-scripts, "area(w, h)"]
Set Field ["temp", "External("Param Receive", "a)"]
Set Field ["temp", "External("Param Get", "a)"]
```

The script sends two values ($w=5$ and $h=3$) to the sub-script named $area(w, h)$. Line 4 names the returned value. Line 5 fetches this value (now a variable) by name, and places it in the field *temp*. Note that the sub-script is called $area(w, h)$ instead of simply *area*. This is convention only, and recommended as it reminds you what values to supply to the function when calling it. (When scripts become more complex I choose to extend this naming convention to something like $(a)=area(w, h)$, which indicates a single return value also.)

The sub-script $area(w, h)$ looks like:

```
Script area(w, h)
Set Field ["temp", "External("Param Begin", "w, h)"]
Set Field ["temp", "External("Param Get", "w") * External("Param Get", "h)"]
Set Field ["temp", "External("Param Return", "temp)"]
Set Field ["temp", "External("Param End", "")"]
```

The first line names the parameters passed, in the order they are sent, separated by a comma (the space after the comma is optional, and ignored by the plugin). Within the sub-script, the parameters can be obtained by name as many times as required, until *Param End* is called. On the second line the named parameters are multiplied together, and the result placed into *temp*.

On the third line the result (in *temp*) is provided as the value to be returned to the calling script³. Before returning, the script *must* call *Param End* to clean up. It disposes of the parameters w and h , and ensures that the returned value is made available to the calling script.

Note: sub-scripts may return more than one value by making repeated calls to *Param Result* (once for each value to be returned). In this case the caller receives the multiple values by providing a list of names (comma delimited) to *Param Receive*. Eg:

```
Set Field ["temp", "External("Param Receive", "result1, result2)"]
```

The values can then be obtained by name, using *Param Get* in the usual manner.

◆ Example 3 — Using local variables

To be written.

◆ Example 4 — Putting it all together — Recursion

To be written. (Function factorial).

³ Lines 2 & 3 could have been combined into a single script step. They are deliberately separated here for clarity.

Function Summary

◆ Param Version

This function serves two purposes. With no argument (the empty string) the function simply returns a string describing the plugin, its registration status (if applicable) and the version number of the plugin. For example:

```
Set Field ["temp", "External("Param Version", "")"]
```

will return a string similar to "Jazz Media Param Routines 1.00". The 'TextToNum' function can be used to return the version number on its own (as a decimal number), from which comparisons can be made. (For this reason the version number will contain only a single decimal point, eg "1.21" and not "1.2.1", as has become common practise today). The following example places the numeric version in the field 'plugin version'.

```
Set Field ["plugin version", "TextToNum(External("Param Version", ""))"]
```

The second purpose of this function is in registration. The serial number provided to registered users should be given to this function when a database (which uses this plugin) is opened. The returned string indicates whether registration was successful or not. For example, if your serial number is "12345":

```
Set Field ["temp", "External("Param Version", 12345)"]
```

sets 'temp' to either "Jazz Media Param Routines 1.00" or "Jazz Media Param Routines 1.00 - Unregistered".

Even though the registration code need only be run once to register use for all open databases, it is recommended that a script containing the registration code is placed in the start-up script for every database which uses this plugin (enabled in FileMaker's document preferences).

If you accidentally perform a script with a (non-empty) invalid serial number, you will need to quit and re-launch FileMaker before the correct serial number will be accepted. You can check your registration status at any time by providing an empty string to this function, as shown earlier.

◆ Param Send

This function sends a single value to a sub-script. The function may be called more than once, to send multiple values. This function should be used immediately prior to calling a sub-script. The format of the command is:

```
Set Field [<field>, "External("Param Send", <value>)"]
```

The following example passes two parameters (values) to the script called *do something*. The first parameter is the string "ABC" and the second is a FileMaker field called 'user name':

```
Set Field ["temp", "External("Param Push", "ABC")"]  
Set Field ["temp", "External("Param Push", user name)"]
```

```
PerformScript[Sub-scripts, "do something"]
...
```

◆ Param Receive

This function should be used after calling a sub-script (when the sub-script has returned), to accept and name the returned values. The function arguments should be a comma-separated list of variable names. The number of names *must* be identical to the number of values returned from the script.

Usage:

```
Set Field [<field>, "External("Param Receive", "<name, ...>")"]
```

If a sub-script *circle area* returns the area of a circle, the value is received and named following the sub-script call, as shown in the following example:

```
...
PerformScript[Sub-scripts, "circle area"]
Set Field ["temp", "External("Param Receive", "area")"]
```

If a sub-script *circle size* returns two values: the area and the circumference (in that order), the above example is modified as followed:

```
...
PerformScript[Sub-scripts, "circle size"]
Set Field ["temp", "External("Param Receive", "area, circumference")"]
```

This function receives and names the values returned from a sub-script, and must be called immediately after calling (and returning from) a sub-script. While you must receive the values immediately, you need not use them straight away. Call *Param Get* with the appropriate value's name to obtain its value at any time after *Param Receive* has been called.

Param Receive must only be called once after the sub-script, but *Param Get* may be called multiple times, as required. See *Param Get* for information on retrieving named values.

Names should start with a letter, followed by any combination of letters, numbers, spaces and underscores, up to 31 characters in length, and not ending with a space. Although some of these restrictions are not presently enforced, adherence ensures compatibility with future enhancements to the plugin. Names are case-sensitive.

◆ Param Begin

This function should be used at the start of a sub-script, to label the parameters passed to it by calling scripts. The function arguments should be a comma-separated list of variable names (labels). The number of labels *must* be identical to the number of values sent to this sub-script by the calling script.

Usage and three separate examples:

```
Set Field [<field>, "External("Param Begin", "<label, ...>")"]
Set Field ["temp", "External("Param Begin", "")"]
```

```
Set Field ["temp", "External("Param Begin", "title)"]
Set Field ["temp", "External("Param Begin", "width, height)"]
```

Calls to *Param Begin* must be paired with calls to *Param End*, the latter should be called just before returning from the same sub-script. See the previous chapter for examples of typical use.

Param Begin creates a new *scope* for local variables. The *scope* ends at the matching call to *Param End*. This means that all variables created with *Param Local* are defined only within the enclosed pair of *Param Begin* and *Param End* calls. (ie. *Param Begin* and *Param End* define the *scope* for local variables).

All parameter names should start with a letter, followed by any combination of letters, numbers, spaces and underscores, up to 31 characters in length, not ending with a space. Although some of these restrictions are not presently enforced, adherence ensures compatibility with future enhancements to the plugin. Names are case-sensitive.

◆ Param End

This function should be called just before returning from a sub-script which receives parameters from or returns results to the calling script.

Usage and example:

```
Set Field [<field>, "External("Param End", "")"]
Set Field ["temp", "External("Param End", "")"]
```

This routine should be paired with every *Param Begin* call. *Param End* deletes all local variables created (variables created since the most recent stack frame) and any parameters named in the *Param Begin* call. Additionally, *Param End* prepares any returned values to be received by the calling routine.

◆ Param Get

Retrieves the named parameter or local variable. The format of the command, followed by an example are:

```
Set Field [<field>, "External("Param Get", "<name>")"]
Set Field ["user name", "External("Param Get", "my name)"]
```

The example places the parameter or local variable previously saved under the name 'my name' into the *FileMaker* field called 'user name'.

If more than one parameter or local variable has been created with the same name (using *Param Begin*, *Param Set*, *Param Local* or *Param Receive*), the most recently created entity will be retrieved. Note: there is no difference between parameters and variables, other than the circumstances under which they are created.

◆ Param Set

This function sets the value of an existing variable (or parameter). If a local

variable does not exist by the name provided, then the most recently created entity by this name will be used — which could be a local variable or a parameter of the parent script (the script which called this script), or of its parent’s parent, and so on. As a last resort, if no entity exists by the name provided, then a local variable will be created.

The value set be retrieved by name using the *Param Get* function.

Usage and example:

```
Set Field [<field>, "External("Param Set", "<name>=<value>")"]  
Set Field ["temp", "External("Param Set", "person=fred)"]
```

A space before the equals sign is allowed, and will not be taken as part of the name. Any spaces after the equals sign are assumed to be part of the value.

Names should start with a letter, followed by any combination of letters, numbers, spaces and underscores, up to 31 characters in length, not ending with a space. Although some of these restrictions are not presently enforced, adherence ensures compatibility with future enhancements to the plugin. Names are case-sensitive.

Note that no distinction is made between variable names and parameter names. If you provide the name of a parameter then this function will set it to the new value. It is unlikely that you would choose to use the same names anyway, as *Param Get* fetches by name only.

Param Set differs from *Param Local*, because it does not guarantee that the variable will be local to this sub-script. See *Param Local* for further information about the differences. In most cases you should use *Param Local* unless you are sure the variable or parameter already exists.

◆ Param Local

This function creates a local variable of the given name and value. If a local variable already exists by this name then it is reused. The value can be retrieved by name using the *Param Get* function.

Usage and example:

```
Set Field [<field>, "External("Param Local", "<name>=<value>")"]  
Set Field ["temp", "External("Param Local", "person=fred)"]
```

A space before the equals sign is allowed, and will not be taken as part of the name. Any spaces after the equals sign are assumed to be part of the value.

Names should start with a letter, followed by any combination of letters, numbers, spaces and underscores, up to 31 characters in length, not ending with a space. Although some of these restrictions are not presently enforced, adherence ensures compatibility with future enhancements to the plugin. Names are case-sensitive.

Param Local differs from *Param Set*, because *Param Local* guarantees the variable will be local to this sub-script. It will not overwrite a variable of the same name from another sub-script. A local variable is always created (unless it already exists, in which case this variable is reused). This behaviour is unlike *Param Set*,

which will use (and overwrite) the most recent variable created, which may or may not be local to this sub-script.

If you are unsure whether to use *Param Local* or *Param Set*, follow this simple rule: If you are creating a new variable, use *Param Local*. If you are setting the value of an existing variable, use *Param Set*.

For advanced users, the *scope* of a local variable is defined as within the paired *Param Begin* and *Param End* calls.

Calls to *Param Begin* should be balanced with calls to *Param End*, and all local variables created inside this pair are available only until *Param End* is called. *Param Local* will not overwrite any variables of the same name created prior to a call to *Param Begin*. See *Param Begin* for further details, and an example of use.

◆ Param Clear All

This function clears all variables and parameters of the local script, and all parent scripts (scripts which called this one). This functions should only be used prior to aborting a script.

Example:

```
Set Field ["temp", "External("Param Clear", "")"]  
Halt Script
```

This releases all memory used by the plugin for its operation. If you don't call *Param Clear All* when you halt a script, your scripts will still work. Eventually *FileMaker* might run out of memory, and request that you quit and restart the application.

An alternate philosophy is to call *Param Clear All* when the outer-most script commences, ensuring that memory is cleared when it is safe to do so. If you never call *Halt Script* then you should never need to worry about calling *Param Clear All*. The plugin cleans up after itself when used properly.

◆ Param Return

This function accepts values to be returned from a sub-script to its calling script. This function may be called more than once to return multiple values to the calling script.

Usage and example:

```
Set Field [<field>, "External("Param Result", "<value>")"]  
Set Field ["temp", "External("Param Result", 29)"]
```

The following example, the sub-script returns two values: '7' and '21' to the calling script. (In practise a script would return something more meaningful than known values).

```
Sub-script  
Set Field ["temp", "External("Param Begin", "...")"]  
...  
Set Field ["temp", "External("Param Return", "7")"]
```

```
Set Field ["temp", "External("Param Return", "21")"]  
Set Field ["temp", "External("Param End", "")"]
```

How it works

A full explanation is beyond the scope of this documentation. It would require an understanding of stacks and frames, as used by compilers and assembler programmers. Details of the internal operation has deliberately been hidden from the user (as much as is possible) to broaden the plugin's appeal.

In brief, this plugin simulates a processor stack using linked-lists, with enhancements such as naming stack elements and framing. The plugin functions operate on this stack. Operation is modelled off the internal operation of the Motorola 68000 series processors, but would likely apply to many other microprocessors of today.

Appendices

◆ Version History

- 1.06 Released to the public as shareware (June 2002)
- 1.07 Fixed bug where no result was returned when shareware dialog was displayed. Decreased dialog frequency.
- 1.08 Fixed bug where setting a parameter or variable to the empty string failed to replace a previous value.

◆ About Jazz Media

Jazz Media is an Australian-based company, who's main business is in creating MultiMedia products for clients. It specialises in CD-ROMs and dynamic web sites requiring a database back-end. Use of an in-house database engine for CD-ROMs provides extremely fast access to data, and means that clients can distribute solutions royalty free.

In the course of normal business plugins have been developed, either to solve specific developmental problems, or to add functionality to programs and bundled as part of the commercial products. After years of successful operation, these plugins have been re-packaged and released to the public as shareware or freeware.

