



>

Jazz Params Plug-in

>

Updated 4 February, 2007

◇

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 workarounds, such as accessing global fields through relationships, 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 the 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 single 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.

However, earlier versions of this plugin have been used in several thousand commercial CD-ROMs (Mac and Windows) for years prior to its release as shareware. It has proved to be a very valuable and reliable plugin for these products.

Conventions

◆ Definitions

The terms *function* is used to refer to an *external* function in the plugin. 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*. If so, context should make it clear whether an *external* (plugin) function or a *sub-script* is being referenced.

◆ Calling functions

Version 1.5 (and later) of this plugin makes use of the newer *FileMaker Pro* plugin architecture. *Plugins* can take multiple values (arguments) including none, and return a single value. Although some of the functions in this plugin don't need to return any value, they will in fact return an empty string ("") and they still need to be wrapped in a script step as if they did return a value.

Plugins are designed to be used inside *FileMaker* formulas. For example, if a plugin has a function to calculate the square of a number it might be included inside a calculation like:

```
Set Field [Area; 3.1415 * Square(Radius)]
```

This calculates the radius of a circle, where 'Area' and 'Radius' are *FileMaker* fields, and 'Square' is the name of a function. The function *Square* accepts the field 'Radius' as its first and only argument (also known as its first parameter), and returns the square of this number, which is used as a formula in the 'Set Field' function.

When an external function returns a value which you will use, you must call it as shown above. However when an external function does not return a value¹ (or the value is unimportant) there are two ways you might choose to use it. The first is to create an If-End If pair of functions, with no script steps in between. The return value of the function is thus ignored. If an external were to display a dialog box of information, you could call it this way:

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

A more compact way (vertically) to call this function is to save the result into a dummy field. I create a global field called 'temp' which I reserve for off loading unwanted function return values. The same function call using this method would look like:

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

I prefer the second method because of its compactness, and it will be used in the examples which follow. The advantage of the first method is that no dummy field is required.

¹ Actually all external functions return an argument, irrespective of whether it is required or not. If a return value is not required the function will return the empty string ("").

◆ Local variables

You can create local variables inside a sub-script which disappear when the sub-script exists (actually when *ParamEnd* is called). Once created these local variables are accessed the same way as other variables or parameters — with *ParamGet* and *ParamSet*.

You create a local variable *after* you call *ParamBegin* with the function *ParamLocal* with exactly the same syntax as *ParamSet*. However using *ParamLocal* ensures that you do not overwrite a variable outside this script of the same name.

```
Set Field [temp ; ParamLocal(<name>, <value>)]
```

You can call *ParamLocal* anytime after *ParamBegin* and before *ParamEnd*, but it should only be called once. Subsequent access should be with *ParamGet* and *ParamSet*. It is good practise to call it very early on in your sub-script, even if you set it to zero or an empty string (""), This ensures *ParamLocal* isn't called more than once for any local variable name you choose. You may of course call *ParamLocal* more than once to created different variables.

Technical note: calling *ParamLocal* more than once for the same local variable name will still work, it is just memory inefficient, as it creates a new variable every time. *ParamGet* and *ParamSet* will always access the most recently created variable.

An example is still to be written.

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 ; ParamSend("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 ; ParamBegin("title")]
Set Field [temp ; ParamGet("title")]
...
Set Field [temp ; ParamEnd]

```

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 ; ParamSend(5, 3)]
PerformScript[Sub-scripts, "area(w, h)"]
Set Field [temp ; ParamReceive("a")]
Set Field [temp ; ParamGet("a")]

```

The script sends two values (5 and 3) to the sub-script named *area(w, h)*. The sub-script returns a single value which we receive and name it 'a'. Line 4 fetches this value (now a local 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 ; ParamBegin("w, h")]
Set Field [area ; ParamGet("w") * ParamGet("h")]
Set Field [temp ; ParamReturn(area)]
Set Field [temp ; ParamEnd]

```

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)². Within the sub-script, the parameters can be obtained by name as many times as required, until *ParamEnd* is called. On the second line the named parameters are multiplied together, and the result placed into a global field called *area*.

² Alternatively they can be provided as separate arguments, such as: `ParamBegin("w"; "h")` which is arguably more elegant and certainly more flexible, but also more cumbersome typing.

On the third line the result (in *area*) is provided as the value to be returned to the calling script. 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.

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

Note: sub-scripts may return more than one value. See the documentation for *ParamReturn* for further details.

◆ Example 3 — Putting it all together (Recursion)

To be written. (Function factorial).

Function Summary

◆ ParamVersion

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

```
ParamVersion
```

will return a string similar to "Jazz Params Version 1.50". 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(ParamVersion)]
```

The second purpose of this function is for 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 ; ParamVersion(12345)]
```

sets 'temp' to either "Jazz Params Version 1.50" or "Jazz Params Version 1.50 - 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 calling this function without an argument, as shown earlier.

◆ ParamSend

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>, ParamSend(<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 ; ParamSend("ABC")]
Set Field [temp ; ParamSend(user name)]
PerformScript[Sub-scripts, "do something"]
...
```

◆ ParamReceive

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 can be provided as separate arguments to *ParamReceive* or can be a single comma-separated list of variable names³. The number of names *must* be identical to the number of values returned from the script.

Usage:

```
Set Field [<field>, ParamReceive("<name>"; ...)]
Set Field [<field>, ParamReceive("<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 ; ParamReceive("area")]
```

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

```
...
PerformScript[Sub-scripts, "circle size"]
Set Field [temp ; ParamReceive("area"; "circumference")]
```

or:

```
PerformScript[Sub-scripts, "circle size"]
Set Field [temp ; ParamReceive("area, circumference")]
```

³ The latter is easier on the typing requirements, but is also partially for backward compatibility with earlier versions.

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 *ParamGet* with the appropriate value's name to obtain its value at any time after *ParamReceive* has been called.

ParamReceive must only be called once after the sub-script, but *ParamGet* may be called multiple times, as required. See *ParamGet* 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, and unicode aware.

◆ ParamBegin

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 can be provided as separate arguments to *ParamBegin* or can be a single 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 four separate examples:

```
Set Field [<field>, ParamBegin", "<label>"; ...]
Set Field [temp ; ParamBegin", "<label, ...>"]

Set Field [temp ; ParamBegin", ""]
Set Field [temp ; ParamBegin", "title")
Set Field [temp ; ParamBegin", "width"; "height")
Set Field [temp ; ParamBegin", "width, height")]
```

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

ParamBegin creates a new *scope* for local variables. The *scope* ends at the matching call to *ParamEnd*. This means that all variables created with *ParamLocal* are defined only within the enclosed pair of *ParamBegin* and *ParamEnd* calls. (ie. *ParamBegin* and *ParamEnd* 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 and unicode aware.

◆ ParamEnd

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>, ParamEnd]
Set Field [temp ; ParamEnd]
```

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

◆ ParamGet

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

```
Set Field [<field>, ParamGet("<name>")]
Set Field ["user name", ParamGet("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 *ParamBegin*, *ParamSet*, *ParamLocal* or *ParamReceive*), 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.

◆ ParamSet

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 *ParamGet* function.

Usage and example:

```
Set Field [<field>, ParamSet("<name>"; "<value>")]
Set Field [temp ; ParamSet("person"; "fred")]
```

Note that the syntax has changed in this function. The previous syntax which used an 'equals' sign is no longer allowed.

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 *ParamGet* fetches by name only.

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

◆ ParamLocal

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 *ParamGet* function.

Usage and example:

```
Set Field [<field>, ParamLocal("<name>"; "<value>")]
Set Field [temp ; ParamLocal("person"; "fred")]
```

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.

ParamLocal differs from *ParamSet*, because *ParamLocal* 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 *ParamSet*, 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 *ParamLocal* or *ParamSet*, follow this simple rule: If you are creating a new variable, use *ParamLocal*. If you are setting the value of an existing variable, use *ParamSet*.

For advanced users, the *scope* of a local variable is defined as within the paired *ParamBegin* and *ParamEnd* calls.

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

◆ ParamClearAll

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 ; ParamClear]
Halt Script
```

This releases all memory used by the plugin for its operation. If you don't call *ParamClearAll* 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 *ParamClearAll* when the outermost 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 *ParamClearAll*. The plugin cleans up after itself when used properly.

◆ ParamReturn

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>, ParamResult("<value>")]
Set Field [temp ; ParamResult(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 ; ParamBegin]
...
Set Field [temp ; ParamReturn(7; 21)]
Set Field [temp ; ParamEnd]
```

You may, if you wish, make repeated calls to *ParamReturn* for each individual return value, as was required with previous versions of this plugin.

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 from 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.
- 1.50 Reworked plugin for Intel Macs, using newer plugin architecture. Added unicode compatibility. Mac release (Feb 2007).
- 1.51 Fixed a bug where the shareware reminder caused an incorrect result to be returned. (Feb 2007)

◆ 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 repackaged and released to the public as shareware or freeware.