



>

## Jazz Stack Plug-in

>

◇

Updated 4 February, 2007

---

## Introduction

### ◆ Overview

This plugin provides a mechanism whereby values can be pushed and popped to and from a stack. Other functions allow the stack to be peeked, to be used as a pipe (allows pulling values from the other end) and marking of stack locations for quick “unwinding”.

The stack is available globally; that is, every opened database accesses the same stack. A typical use could be to implement simple “go back” functionality, by recording the history of records a user views, by diligently placing the appropriate record IDs on the stack.

### ◆ Shareware or Freeware

This plug-in is shareware. It will remind you of this fact periodically unless you register. Registration does not cost much, and encourages me to write more plugins. If you use this plug-in regularly, please register it at <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 warranty of any kind. You are responsible for determining its 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 value<sup>1</sup> (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.

---

<sup>1</sup> 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 ("" ).

## General use

### ◆ Simple stack use

A stack is also known as a *FIFO* buffer — first in last out. Whatever you add to the stack is retrieved in reverse order. So if you *push* 1, 2, 3 (in separate operations, in that order) then consecutive *pops* will return 3, 2, 1.

The stack is considered to grow upwards, so the last element placed on the stack is called the top of the stack (*TOS*). All new elements *pushed* are placed on top of the existing elements in the stack. So the top of the stack keeps changing.

For example:

Set Field [temp ; StackPush(1)]	-- stack contains: 1 (TOS=1)
Set Field [temp ; StackPush(2)]	-- stack contains: 1, 2 (TOS=2)
Set Field [top ; StackPeek(0)]*	-- places 2 (the top of the stack) into field <i>top</i> .
	-- the stack is unaffected (still contains 1, 2)
Set Field [temp ; StackPush(3)]	-- stack contains: 1, 2, 3 (TOS=3)
Set Field [temp ; StackDrop]	-- removes 3 from the top of the stack) into temp.
	-- stack contains 1, 2 (TOS=2)
Set Field [temp ; StackDrop]	-- removes 2 from the top of the stack) into temp.
	-- stack contains 1 (TOS=1)
Set Field [temp ; StackDrop]	-- removes 1 from the top of the stack) into temp.
	-- stack is now empty

\*The *StackPeek(0)* function is not necessary for the above stack manipulation. It demonstrates how you can view the top of the stack without affecting it. You might choose to call this at times when you are experimenting with stacks, until you are comfortable with their operation.

### ◆ Marking the stack and dropping

With simple stacks you might only add one item and remove one item from the stack for an operation. However with more complex stacks you could add a number of items to be remembered for a single operation. You must then be careful that you never remove more or less items than you are meant to — than was placed there for removal. If you do, the stack will become out of sync, and useless.

To make life easier, you can place a *mark* on the stack. A mark is simply a name you give to that exact position on the stack. If you ever wish to return to that point on the stack without knowing how many items you have added since, you may simply drop the stack to that mark.

### ◆ Example — Implementing “go back” functionality

A common use for a stack is to create a go-back history, similar to how a web browser works. Every time you leave a location you place enough information on the stack to enable you to return (to know where to return to). The example on the

web site implements a simple database of PHP maths functions.

In this example, when you click a function in the 'see also' list, a script places the record ID on the stack. When you wish to go back, a record ID is retrieved from the stack, and the script takes you to that record ID. This simulates go-back functionality.

In the interests of simplicity the example is not perfect, as it doesn't remember the previous found set, just the single record you were viewing. Download the example and examine the scripts to see how this is implemented.

---

## Function Summary

### ◆ StackVersion

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:

```
StackVersion
```

will return a string similar to "Jazz Stack 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(StackVersion)]
```

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 ; StackVersion(12345)]
```

sets 'temp' to either "Jazz Stack Version 1.50" or "Jazz Stack 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.

## ◆ StackPush

This function pushes a value onto the top of the stack. The format of the command, followed by two examples of use are:

```
Set Field [temp ; StackPush( <value>)]  
Set Field [temp ; StackPush("ABC")]  
Set Field [temp ; StackPush( user name)]
```

In the first example, the string "ABC" is pushed onto the stack. In the second example the contents of the FileMaker field called 'user name' is pushed onto the stack.

For convenience the function returns the value of the item pushed, otherwise it returns the empty string (eg. if not enough memory is available to save the value).

## ◆ StackPop

This function removes and returns the topmost element from the stack. That is, the most recent value placed on the stack with a *Stack Push* command is removed from the stack and returned to the user. The format of the command, followed by an example is shown below:

```
Set Field [<field>, StackPop]  
Set Field ["name", StackPop]
```

The example places the popped value into the FileMaker field called 'name'. If there is no value on the stack (the stack is empty) then the empty string will be returned.

## ◆ StackPull

This function removes the bottom-most element from the stack. In conjunction with the *StackPush* command, a pipe can be simulated (whereby the first element pushed on the stack is the first to be removed, or *pulled*). The format of the command, followed by an example is shown below:

```
Set Field [<field>, StackPull]  
Set Field ["name", StackPull]
```

The example places the pulled value into the FileMaker field called 'name'. If there is no value on the stack (the stack is empty) then the empty string will be returned.

## ◆ StackPeek

This function returns the value at the location on the stack as specified by *index*. The format of this command is:

```
Set Field [temp ; StackPeek( <index>)]
```

The stack is unaltered by this command. If *index* is 0, the top of the stack is returned. If *index* is 1, the element second from the top of the stack is returned, and so on. If the element specified does not exist, the empty string is returned.

### ◆ **StackSize**

This function returns the size of the stack (the count of elements on the stack). An empty stack returns 0. The following example places the stack size into the FileMaker field 'temp':

```
Set Field [temp ; StackSize]
```

### ◆ **StackMark**

This function places a mark next to the current top of stack. (You can use any name as the mark). Placing a mark does not affect the size or contents of the stack in any way, it simply gives an element a name, and flags it as *marked*. The mark will follow the element as the stack grows and shrinks. The mark is lost when the element is removed from the stack.

Use this function when you wish to remember the current top-of-stack for later, in combination with the *StackGetMark* and *StackDropToMark* routines.

Usage and example:

```
Set Field [temp ; StackMark("<name>")]  
Set Field [temp ; StackMark("fred")]
```

Calls to *StackMark* and *StackDropToMark* are often paired. The latter lets you quickly return to the stack as-it-was when it was previously marked.

### ◆ **StackGetMark**

This function returns the name of the mark applied to the stack at the location specified by *index*. This function is similar to *StackPeek*, except that it returns the mark placed at that stack element (if any) rather than the value.

The format of this command is:

```
Set Field [temp ; StackGet Mark(<index>)]
```

The stack is unaltered by this command. If *index* is 0, the mark at the top of the stack (if any) is returned. If *index* is 1, the mark of the element second from the top of the stack is returned, and so on. If there is no mark at the specified element, or the element does not exist (the index is bigger than the stack), the empty string is returned.

## ◆ StackDrop

This command removes the top N elements from the stack. It is similar to calling *StackPop* several times, and discarding the values. The function returns *true* (1) if an item existed to drop, or *false* (0) if the full requested number of elements could be dropped (leaving the stack empty).

Usage and example:

```
Set Field [temp ; StackDrop(<count>)]  
Set Field [temp ; StackDrop(2)]
```

The example drops two elements from the top of the stack.

## ◆ StackDropToMark

This function drops elements from the top of the stack, up to but not including the element containing the mark of the given name.

If more than one mark exists with this name, the first (topmost) location will be used. If a mark by the given name can not be found anywhere on the stack, the function does nothing, and returns *false* (0). Names are case-sensitive.

Usage:

```
Set Field [temp ; StackDropToMark("<name>")]  
Set Field [temp ; StackDropToMark("fred")]
```

## ◆ StackClear

This routine clears the entire stack.

Usage and example:

```
Set Field [temp ; StackClear]
```

---

## ▶ How it works

This plugin is implemented by creating a linked list of elements, operating as a stack. Each element can contain a value and a name (a string of up to 31 characters). Named elements are said to be *marked*.

More to be written.

## Appendices

### ◆ Version History

- 1.06 Released to the public as shareware (June 2002)
- 1.07 Fixed bug where no result was returned when the shareware dialog was displayed. Decreased dialog frequency.  
Corrected version string version number in *Stack Version* function.
- 1.08 Internal code base update only. Issues with previous code base are not believed to impact on *Jazz Stack*.
- 1.50 Reworked plugin for Intel Macs, using newer plugin architecture. Added unicode compatibility. Mac release (Feb2007).
- 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.