Version 1 (2016-11-16)

LiveCode for FM Handbook

Introduction

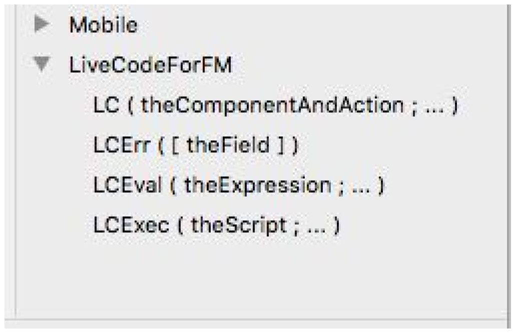
LiveCode for FM (LCFM) allows you to tap into the full power of LiveCode to use in your FileMaker solutions.

By using the LCFM Workspace (workspace) you can define custom functions using the LiveCode Script programming language, create simple forms from a selection of control types and add custom windows by building content in the LiveCode IDE. Each of these individual pieces of functionality is called a *component*.

In this document we will describe how to use LCFM functionality in your FileMaker solutions, how to interact with FileMaker in your LiveCode scripts and how to make a stack (custom window) accessible from FileMaker.

Plugin Functions

The LiveCode for FM plugin provides a small number of functions you can include in your solutions via the Specify Calculation dialog - you can find these functions under *LiveCodeForFM* in the sidebar:


▶ Mobile
▼ LiveCodeForFM
LC (theComponentAndAction ; ...)
LCErr ([theField])
LCEval (theExpression ; ...)
LCExec (theScript ; ...)

Most of the functionality provided by LCFM is accessed using the generic *LC* plugin function - this allows you to call specific components, whether they be built-in, or defined by you.

The *LCEval* and *LCExec* plugin functions allow you to evaluate a LiveCode Script expression, or execute a sequence of LiveCode Script statements directly.

The *LCErr* plugin function allows you to check any error state arising from the most recent use of *LC*, *LCEval* or *LCExec*.

Checking for Errors - LCErr

If an error occurs whilst running one of the LC functions (LC, LCEval or LCExec), information will be made available to FileMaker via the *LCErr* function:

```
LCErr ( [ theField ] )
```

The LCErr function takes a single optional parameter. If the parameter is not present, then LCErr will return true if the previous LC function caused an error, or false if it did not cause an error. You can use this to check success or failure of an LCFM action in your FileMaker scripts.

You can pass a parameter to LCErr to get more information about the error that occurred. The *theField* parameter can be one of: "type", "reason", "component", "action", "line" or "column".

Type

The **type** field indicates what kind of error occurred.

If the type is "compile", then it indicates that there was a syntax error in a script which was executed. For example attempting to evaluate the LiveCode Script expression "x is - not y" will cause a compile error, because it is not valid LiveCode Script syntax.

If the type is "runtime", then it indicates there was an error whilst executing a script. For example, attempting to evaluate the LiveCode Script expression "1 + a" will cause a runtime error because you cannot add 1 to the letter a.

If the type is "error", then it indicates that a script explicitly raised an error. This can happen either through use of the LiveCode **throw** command, or by returning an error value from a handler you have defined.
If the type is "toolbox", then it indicates that an error occurred whilst trying to access a component. For example, calling LC("myComponent") from FileMaker will cause a toolbox error if there isn't a component called *myComponent*.

Reason

The **reason** field gives a human-readable description of the error that occurred. In the case of compile, runtime and toolbox type errors these are pre-defined strings explaining the error. In the case of error type errors, this will be the string you passed to the **throw** command, or returned in the error value.

Component / Action

The **component** and **action** fields indicate what component caused the error, and what was being done when it happened. These fields will only be present if the error occurred whilst calling *LC*. In this case, the component field will be the name of the component being called, and the action will be the action which was selected.

Line / Column

The **line**, and **column** fields indicate where the error occurred when using *LCEval*, *LCExec* or *LC* (if calling a function component). They are the line and column of the piece of script which caused the error.

Evaluating an Expression - LCEval

You can evaluate an arbitrary LiveCode Script expression by using *LCEval*:

```
LCEval ( theExpression ; ... )
```

For example, LCEval("1 + 2") will return (the number) 3 and LCEval("the version") will return the LiveCode engine's version string.

Any extra parameters you pass to LCEval after *theExpression* can be accessed using the *fmParam()* or *fmNativeParam()* FileMaker API functions.

For example, LCEval("fmParam(1) + fmParam(2)"; 100; 200) will return 300.

If evaluating the expression causes an error, then the invalid value will be returned, and you can find out what went wrong by using the LCErr function.

Executing Statements - LCExec

You can execute a sequence of LiveCode Script commands by using *LCExec*:

```
LCExec ( theCommands ; ... )
```

For example, LCExec("return 1 + 2") will return (the number) 3.

To execute a sequence of commands, separate them by ';' or by LF (ascii 10). For example, LCExec("put 1 + 3 into tempVar; return tempVar") will also return (the number) 3.

Similar to LCEval, you can pass extra parameters to the LCExec call, and access them using the *fmParam()* or *fmNativeParam()* FileMaker API functions.

For example, LCExec("if fmParam(1) then return 1 else return 2"; true) will return 1.

If executing the statements causes an error, then the invalid value will be returned, and you can find out what went wrong by using the LCErr function.

Calling Components - LC

You can call a component, whether built-in, or one you have defined by using *LC*:

```
LC ( theComponentAndAction ; ... )
```

The *theComponentAndAction* parameter should be a string of the form "<component-name>" or "<component-name>.<action>".

Here the parameters you pass are defined by the component and action you choose. We explain how to use the different component types in the next section.

If calling the component causes an error, then the invalid value will be returned and you can find out what went wrong by using the LCErr function.

Components

LiveCode functionality is made available to FileMaker through *components*. LCFM ships with a number of built-in ones, but you can also define your own through the workspace.

The built-in components are shipped within the LCFM plugin, and are not editable. If you wish to modify one, or inspect it then you can them in the workspace and edit the copy.

Components you define yourself can be found through Finder at *Documents/My LiveCode/FileMaker Components*.

Note: If you use LiveCode already, or have installed the LiveCode IDE you can change the location of the *My LiveCode* folder via the LiveCode preferences dialog.

All components must be given a unique name. A component name may consist of any of the characters a-z, A-Z, underscore ('_') or digits. However, you may not define component yourself which starts with the letters 'LC', these are reserved for built-in components.

Note: Component names are always treated case-insensitively - for example, MyComponent and myComponent are treated as the same name.

A component can be a function, a form or 'custom'.

A function component is a small piece of LiveCode Script which is run when the component is called.

A form component is a simple dialog box, composed of a collection of controls. Forms are shown modally when they are called, waiting for the user to close them.

A custom component can be any LiveCode stack (which you can create in the LiveCode IDE). Custom components define their own actions by providing a special *FileMakerAction* handler in their stack script.

Using Functions

Function components can be used via the *LC* plugin function.

Functions require no action, and calling them executes the script defined for them in the workspace.

For example:

```
Set Variable [ $doesitexist ; Value: LC ( "lcFileExists" ; "~/Desktop/MyFile.txt" ) ]
```

Will set the $doesitexist variable to true if there is a file "MyFile.txt" on the user's desktop.

Using Forms

Form components can be used via the *LC* plugin function.

Forms understand three actions *Run*, *Set* and *Get*.

The *Set* action allows you to set the values in the fields defined on the form before showing it:

```
LC ( "<component-name>.Set" ; <field-index> ; <field-value> )
```

The *field-index* is the numeric index of the field on the form (including Text Display fields) and the *field-value* is the value to set it to.

The *Get* action allows you to fetch the values the user chose after showing it:

```
LC ( "<component-name>.Get" ; <field-index> )
```

The *field-index* is the numeric index of the field on the form (including Text Display fields) that you wish to fetch. The function returns the value of that field.

The *Run* action allows you to show the dialog:

```
LC ( "<component-name>.Run" )
```

It returns the label of the button which the user clicked to close the form.

For example, to use the lcRating component you might do the following in a FileMaker Script:

```
Set Variable [ $ignore ; Value: LC ( "lcRating.Set" ; 2 ; 5 ) ]
Set Variable [ $ignore ; Value: LC ( "lcRating.Run" ) ]
Set Variable [ $value ; Value: LC ( "lcRating.Get" ; 2 ) ]
```

Here the $value variable will be set to the value the user chose on the rating's slider.

The values of the fields you can fetch using the *Get* action remain available until the next time you *Run* the same component.

Using Custom Components

Custom components can be used via the *LC* plugin function.

Each custom component can define its own actions in its *FileMakerAction* handler.

For example, to use the built-in camera capture component you might do:

```
Set Field [ MyTable::MyContainer ; LC("lcCameraCapture") ]
```

This will prompt the user to take a picture, which will then be placed in the MyContainer field in MyTable.

Defining Custom Components

A custom component is any LiveCode stack which has been created in the LiveCode IDE, which contains a *FileMakerAction* handler in its stack script.

Custom components can be added to your LCFM workspace through the custom components pane.

We recommend the following approach:

- Create your stack in the LiveCode IDE
- Save the stack
- Close and remove the stack from memory
- Create a new custom component in the LCFM workspace
- Choose the stack you just saved
- Click the Edit button to open Finder to the folder containing the stack
- Load that copy into the LiveCode IDE

In order to make your LiveCode stack into a custom component, callable from FileMaker you should add a *FileMakerAction* handler its stack script:

```
on FileMakerAction pAction
    -- Decide what to do based on pAction
end FileMakerAction
```

The FileMakerAction handler is called whenever the component is called via *LC*.

If *pAction* is empty, then the component should perform a default action (if appropriate); otherwise it should perform the named action.

In addition to the Action handler, custom components may also define Load and Unload handlers:

```
on FileMakerLoad
    -- load any resources and set up any state you need
end FileMakerLoad

on FileMakerUnload
    -- unload all resources you loaded and tear down any state you set up
end FileMakerUnload
```

The Load handler is called when LCFM loads the components on startup, and the Unload handler is called when LCFM unloads the components on shutdown.

Additionally, these handlers will be called when you click the *Reload* button in the workspace. The Reload feature allows you to edit your custom components in a LiveCode IDE, and check the changes immediately in FileMaker without having to restart FileMaker.

Note: Remember to save your custom component stacks in the LiveCode IDE before trying to Reload in FileMaker!

Important: If you use any FileMaker interoperation functions in your custom components then they will currently not work in the LiveCode IDE. You need to test your components in FileMaker.

FileMaker Interoperation

Values

FileMaker, natively, has a number of different types of value which can be manipulated:

- invalid
- text
- number
- binary data
- date / time / timestamp
- boolean

There is also a special error value type unique to LCFM.
Each of these can be manipulated in LiveCode - the simpler ones mapping directly to native values in LiveCode Script.

Values which map simply require no special handling in your scripts; values which don't map natively require the manipulation of specifically structure arrays to handle.

Some FileMaker interoperation functions are available in two forms - native and normal. The native forms return values from FileMaker in their native (array) representation, whilst the normal ones attempt to map the FileMaker value to a LiveCode script value.

Invalid

The invalid value represents an invalid value in FileMaker.

When mapped directly to a LiveCode value, the invalid FileMaker value will appear the empty string (**empty**).

When using the native representation, the invalid value will appear as an array:

```
        tValue["type"] := "invalid"
```

The invalid value can be created in LiveCode Script using the *fmInvalidValue()* function.

Text

The text value represents a unicode string in FileMaker.

When mapped directly to a LiveCode value, text values will appear as LiveCode strings.

When using the native representation, text values will appear as an array:

```
        tValue["type"] := "text"
        tValue["value"] := <value as string>
```

Number

The number value represents fixed precision (decimal) numbers in FileMaker.

When mapped directly to a LiveCode value, number values will appear as LiveCode numbers.

When using the native representation, a number value will appear as an array:

```
        tValue["type"] := "number"
        tValue["precision"] := precision of number
        tValue["value"] := <value as decimal string, or number>
```

Note: LiveCode uses binary floating-point as its numeric representation internally; this means you can lose precision when manipulating numbers coming from FileMaker.

Binary Data

The binary data value represents a collection of named binary streams in FileMaker.

Binary data values never map directly to a LiveCode value, they can only be manipulated via the native representation.

When using the native representation, a binary data value will appear as an array.

```
        tValue["type"] := "binary"
        tValue["main"] := <value of MAIN stream> (if MAIN stream is present)
        tValue["filename"] := <value of FNAM stream> (if FNAM stream is present)
        tValue["width"] := <value of width present in SIZE stream> (if SIZE stream is present)
        tValue["height"] := <value of height present in SIZE stream> (if SIZE stream is present)
        tValue["value"][<stream-1>] := <stream-1 size in bytes> or <stream-1 content>
        ...
        tValue["value"][<stream-n>] := <stream-n size in bytes> or <stream-n content>
```

Here the <stream-i> keys are the four char codes (as four character strings) naming each stream.

When receiving a binary data value from FileMaker (e.g. as the result of the fmNativeParam() function), the named stream keys will contain the size of the data in that stream in bytes.

When sending a binary data value to FileMaker (e.g. by returning from a handler), the named stream keys should contain the actual binary data you want in that stream.

Note: If a MAIN, FNAM or SIZE stream is present, you will not see it in the values array when receiving a binary data value from FileMaker; and if you specify a main, filename, or width and height key when sending a binary data value to FileMaker you should not include them in the value array.

Date / Time / Timestamp

The date, time and timestamp values represent a date a time or a date and time in FileMaker.

When mapped directly to a LiveCode value, date/time/timestamp values will map to strings (DATE, TIME or DATE TIME).

When using the native representation, the datetime values will appear as arrays:

```
tValue["type"] := "date" | "time" | "timestamp"
tValue["value"]["year"] := <year as number> (if date or timestamp)
tValue["value"]["month"] := <month as number>  (if date or timestamp)
tValue["value"]["day"] := <day as number> (if date or timestamp)
tValue["value"]["hour"] := <hour as number> (if time or timestamp)
tValue["value"]["minute"] := <minute as number> (if time or timestamp)
tValue["value"]["sec"] := <sec as number> (if time or timestamp)
tValue["value"]["usec"] := <usec as number> (if time or timestamp)
```

Boolean

The boolean value represents true or false in FileMaker.

When mapped directly to a LiveCode value, boolean values will map to **true** or **false**.

When using the native representation, a boolean value will appear as an array:

tValue["type"] := "boolean"
tValue["value"] := **true** or **false**

Error

When returning a value to FileMaker from LiveCode Script, you can indicate an error occurred by returning an error value.

An error value is represented as an array:

```
    tValue["type"] := "error"
    tValue["value"] := <reason for error as string>
```

You can create an error value in LiveCode Script using the *fmErrorValue()* function.

Functions

LCFM adds a variety of functions to the LiveCode Script environment when running in FileMaker. These features are designed to allow your LiveCode Script code to interact with FileMaker.

fmInvalidValue()

This call returns an array of the correct structure to indicate an 'invalid value' in FM.

fmErrorValue(pReason)

This call returns an array of the correct structure to indicate an error return from a function.

fmParamCount()

This call returns the number of arguments passed to the plugin function from FM. It is an non-negative integer.

fmParam(pIndex)

This call returns the argument with (1-based) index *pIndex* as a LC value. This performs the untyped mapping described above.

Specifically:
- invalid -> empty
- booleans -> true or false
- numbers -> numeric strings
- text -> strings
- date/time/timestamp -> string in FileMaker date/time format
- binary data -> no mapping (this will throw an error)

fmNativeParamType(pIndex)

This call returns the native (array) representation of the argument with (1-based) index *pIndex*.

The return value is one of invalid, boolean, number, text, date, time, timestamp, binary.

This indicates the actual type of the value coming from FileMaker.

fmNativeParam(pIndex)

This call returns the native (array) representation of the argument with (1-based) index *pIndex*.

It allows faithful access to the contents of a FileMaker value.

Note: Access to the actual data values of streams in binary values is done via *fmBinaryParamHasStream()* and *fmBinaryParamStream()* functions.

fmBinaryParamMain(pIndex)

This call returns the MAIN stream type of the binary argument with (1-based) index pIndex.

If there is no MAIN stream, empty is returned.

fmBinaryParamFilename(pIndex)

This call returns the FNAM stream type of the binary argument with (1-based) index pIndex.

If there is no FNAM stream, empty is returned.

fmBinaryParamSize(pIndex)

This call returns "width,height" of the SIZE stream type of the binary argument with (1-based) index pIndex.

If there is no SIZE stream, empty is returned.

fmBinaryParamHasStream(pIndex, pStreamType)

This call returns true if the param with the given (1-based) index is of binary type and has a stream of the given type.

fmBinaryParamStream(pIndex, pStreamType)

This call returns the data contained in the given stream type of argument with the specified (1-based) index.

If there is no stream with the given type, empty is returned.

fmHasCurrentFile()

This call returns true if FileMaker has at least one open solution file.

If there are no open solution files then it returns false.

The fmHasCurrentFile() call has to be true in order to use the *fmEvaluate()*, *fmNativeEvaluate()*, *fmExecuteSQLAsText()* or *fmExecuteSQL()* functions.

fmCurrentFileName()

This call returns the name of the currently active FileMaker solution file.

If there is no solution files open, it returns empty.

The result of this call can be used as the *pFile* parameter in *fmExecuteSQLAsText()* and *fmExecuteSQL()*.

fmEvaluate(pExpression)

This call evaluates the string pExpression as a FileMaker expression in the current (FileMaker) context.

It returns the result of the expression as LiveCode values. If you wish to evaluate a FileMaker expression returning binary data, you must use *fmNativeEvaluate()*.

fmNativeEvaluate(pExpression)

This call evaluates the string pExpression as a FileMaker expression in the current (FileMaker) context.

It returns the resulting value in native (array) representation.

fmExecuteSQLAsText(pSQL, pFile, pParams)

This call evaluates SQL *pSQL* in the context of FileMaker file *pFile* with the specified array of arguments

The *pParams* array should be numerically keyed, starting at one and can be composed of either normal LiveCode values, or native (array) FileMaker values.

The result is a string rendering of result set, using **the columnDelimiter** and **the rowDelimiter** of the caller.

fmExecuteSQL(pSQL, pFile, pParams)

This call evaluates SQL *pSQL* in the context of the FileMaker file *pFile* with the specified array of arguments.

The *pParams* array should be numerically keyed, starting at one and can be composed of either normal values, or native (array) FileMaker values.

The result is a numerically keyed array of numerically keyed arrays. Each value of the secondary arrays are the LiveCode values.

fmNativeExecuteSQL(pSQL, pFile, pParams)

This call evaluates SQL *pSQL* in the context of the FileMaker file *pFile* with the specified array of arguments.

The *pParams* array should be numerically keyed, starting at one and can be composed of either normal values, or native (array) FileMaker values.

The result is a numerically keyed array of numerically keyed arrays. Each value of the secondary arrays are native (array) FileMaker values.