Troi Serial Plug-in 4.5

for FileMaker Pro 15

USER GUIDE

June 2016

Troi Automatisering
Boliviastraat 11
2408 MX Alphen a/d Rijn
The Netherlands

You can also visit the Troi web site at: http://www.troi.com for additional information.

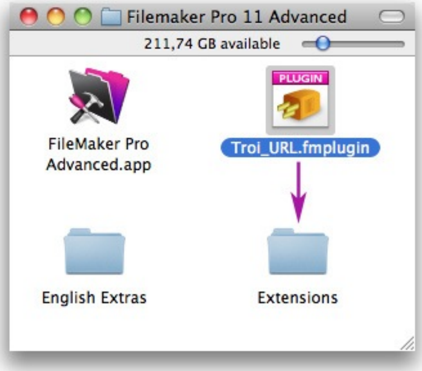
Table of Contents

Installing plug-ins

Starting with FileMaker Pro 12 a plug-in can be installed directly from a container field. Please see the **EasyInstallTroiPlugins.fmp12** example file to install plug-ins with FileMaker Pro 12, 13, 14 and 15. The instructions below are for FileMaker Pro 11.

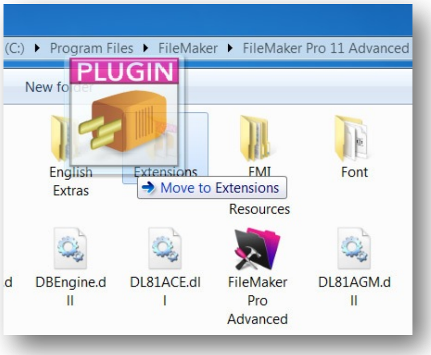
For Mac OS X:

- Quit FileMaker Pro.
- Put the file "Troi_Serial.fmplugin" from the folder "Mac OS Plug-in" into the "Extensions" folder in the FileMaker Pro application folder.
- If you have installed previous versions of this plug-in, you are asked: “An older item named “Troi_Serial.fmplugin” already exists in this location. Do you want to replace it with the one you’re moving?’ Press the OK button.
- Start FileMaker Pro. The first time Troi Serial Plug-in is used it will display a dialog box, indicating that it has loaded and showing the registration status.



For Windows:

- Quit FileMaker Pro.
- Put the file "Troi_Serial.fmx" from the directory "Windows Plug-in" into the "Extensions" subdirectory in the FileMaker Pro application directory.
- If you have installed previous versions of this plug-in, you are asked: “This folder already contains a file called 'Troi_Serial.fmx'. Would you like to replace the existing file with this one?’ Press the Yes button.
- Start FileMaker Pro. The first time Troi Serial Plug-in is used it will display a dialog box, indicating that it has loaded and showing the registration status.



TIP You can check which plug-ins you have loaded by going to the plug-in preferences: Choose **Preferences** from the **Edit** menu, and then choose **Plug-ins**.

You can now open the file "All Serial Examples.fmp12" to see how to use the plug-in's functions. There is also a function overview in the download.

If you have problems

This user guide tries to give you all the information necessary to use this plug-in. So if you have a problem please read this user guide first. You may also visit our support web page:

http://www.troi.com/support/

This page contains FAQ's (Frequently Asked Questions), help on registration and much more. If that doesn't help you can get free support by email. Send your questions to **support@troi.com** with a full explanation of the problem. Also give as much relevant information (version of the plug-in, which platform, version of the operating system, version of FileMaker Pro) as possible.
If you find any mistakes in this manual or have a suggestion please let us know. We appreciate your feedback!

TIP You can get more information on returned error codes from the OSErrrs database on our web site:

http://www.troi.com/software/oserrrs.html

This free FileMaker database lists all error codes for Windows and Mac OS X.

What can this plug-in do?

The Troi Serial Plug-in adds serial functions to FileMaker Pro. With this plug-in you can read and write to any serial port that is available on your computer.

NOTE: USB ports are not supported. USB is a bus protocol that can be used from various purposes and devices, like keyboards, hard disks, CD-ROM drives, adaptors, cameras. All these devices need specific drivers. We have currently no plans to create a USB plug-in. Note however that the Troi Serial Plug-in is reported to be working with the USB to Serial adapters.

Software requirements

System requirements for Mac OS X

Mac OS X 10.6.8 (Snow Leopard), OS X 10.7 (Lion), OS X 10.8 (Mountain Lion), OS X 10.9 (Mavericks) OS X 10.10 (Yosemite), OS X 10.11 (El Capitan).

System requirements for Windows

Windows 7 on Intel-compatible computer 1 GHz or faster.
Windows 8 or Windows 8.1.
Windows 10.

FileMaker Pro requirements

FileMaker Pro 12 or FileMaker Pro Advanced 12.
FileMaker Pro 13 or FileMaker Pro Advanced 13.
FileMaker Pro 14 or FileMaker Pro Advanced 14.
FileMaker Pro 15 or FileMaker Pro Advanced 15.

NOTE We have successfully tested it with FileMaker Pro 11, but we no longer provide active support for this version. Troi Serial Plug-in will also probably run fine with FileMaker 7 to 10, but we have not tested this and we no longer provide support for this.

Troi Serial Plug-in 4.5 will also work with a bound runtime, created with FileMaker Advanced 12, 13, 14 or 15.

FileMaker Server requirements

FileMaker Server 12 or FileMaker Server Advanced 12 or higher.
FileMaker Server 13 or FileMaker Server Advanced 13 or higher.
FileMaker Server 14 or FileMaker Server Advanced 14 or higher.
FileMaker Server 15 or FileMaker Server Advanced 15 or higher.

You can use FileMaker Server to serve databases that use functions of the Troi Serial Plug-in (client-side): you need to have the plug-in installed at the clients that use these functions.

Troi Serial Plug-in can also be used by FileMaker Server as a server-side plug-in or as a plug-in used by the web publishing engine. More information can be found in the download or here:

http://www.troi.com/support/filemaker-server-side-plug-ins.html

FileMaker Server and AutoUpdate

With FileMaker Pro 12 the AutoUpdate feature is no longer needed, as plug-ins can can be installed directly from a container field (See the **EasyInstallTroiPlugins.fmp12** example file to install plug-ins with FileMaker Pro 12 to 15.)

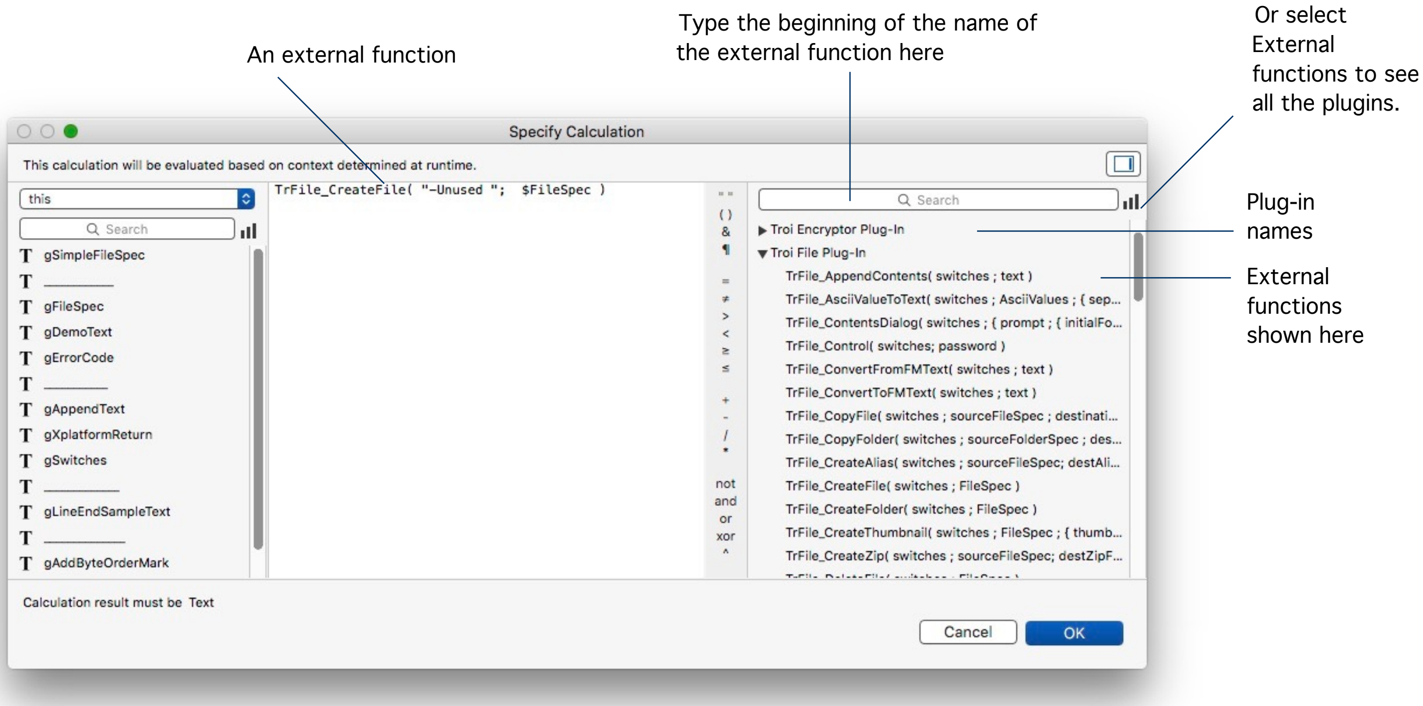
If you still use FileMaker Server 11 you can use the AutoUpdate feature of FileMaker Server 11 to help you automate installing and updating plug-ins automatically. We created an example file and a tar formatted plug-in of Troi Serial Plug-in (only needed on Mac OS X) to get you started. Visit our AutoUpdate web page to download the example:

http://www.troi.com/software/autoupdate.html

Getting started

Using external functions

Troi Serial Plug-in adds new functions to the standard functions that are available in FileMaker Pro. The functions added by a plug-in are called external functions. You can see those extra functions for all plug-ins at the top right of the Specify Calculation box:



An external function

Type the beginning of the name of the external function here

Or select External functions to see all the plugins.

Plug-in names

External functions shown here

This calculation will be evaluated based on context determined at runtime.

```
TrFile_CreateFile( "-Unused ";  $FileSpec )
```

- Troi Encryptor Plug-In
- Troi File Plug-In
  - TrFile_AppendContents(switches ; text)
  - TrFile_AsciiValueToText(switches ; AsciiValues ; { sep...
  - TrFile_ContentsDialog(switches ; { prompt ; { initialFo...
  - TrFile_Control(switches; password)
  - TrFile_ConvertFromFMText(switches ; text)
  - TrFile_ConvertToFMText(switches ; text)
  - TrFile_CopyFile(switches ; sourceFileSpec ; destinati...
  - TrFile_CopyFolder(switches ; sourceFolderSpec ; des...
  - TrFile_CreateAlias(switches ; sourceFileSpec; destAli...
  - TrFile_CreateFile(switches ; FileSpec)
  - TrFile_CreateFolder(switches ; FileSpec)
  - TrFile_CreateThumbnail(switches ; FileSpec ; { thumb...
  - TrFile_CreateZip(switches ; sourceFileSpec; destZipF...

Calculation result must be Text

You use special syntax with external functions: FunctionName(parameter1 ; parameter 2) where FunctionName is the name of an external function. A function can have zero or more parameters. Each parameter is separated by a semi-colon. Plug-ins don't work directly after installation. To access a plug-in function, you need to add the calls to the function in a calculation, for example in a text calculation in Define Fields or in a script.

Where to add the External Functions?

External functions for this plug-in are intended to be used in a Set Field or Set Variable script step using a calculation. For most functions of this plug-in it makes no sense to add them to a define field calculation, as the functions will have side effects. Only the Serial_AsciiValueToText and Serial_TextToAsciiValue functions have no side effects and can be used in a define field calculation.

Simple example

We start with a simple example to get you started. Create a new database, with a global text field called gPortNames. Create a new script called "Get Serial Port Names". Delete all steps and then add the following script step:

```
Set Field[gPortNames, Serial_GetPortNames( "-Unused" ) ]
```

This shows the call to the Serial_GetPortNames function. This function has only one parameter, switches, which is currently not used, so "-Unused" is given as value. Performing this script will return all the serial ports that can be found on this computer, separated by returns.
On Windows the result will be something like this:

```
COM1¶
COM2¶
COM3¶
COM4¶
```

NOTE Function names, likeSerial_GetPortNames, are no longer case sensitive. You can type them or get them from the External Functions list at the top right of the "Specify Calculation" dialog.

Please take a close look at the included example files, as they provide a great starting point. From there you can move on, using the functions of the plug-in as building blocks. Together they give you great new tools to access serial ports!

You can use globals or variables

It is possible to use variables in calculations. Our example files in the download now both use global fields and variables to pass parameters and store the results of a plug-in function.

As this release of Troi Serial is intended for FileMaker Pro 12 and higher, we will use variables wherever possible. Note that not all examples are using variables yet.

All plug-in functions work with variables just fine. For example if you have this script step:

```
Set Field[gErrorCode, Serial_Open( "-Unused" ; “port1" ) ]
```

With variables you can alternative use:

```
Set Variable[$ErrorCode, Serial_Open( "-Unused" ; “port1" ) ]
```

The main advantage of variables is that you don't need to define global fields that clutter your database definitions. The variables can stay local to the script.

Summary of functions

Troi Serial Plug-in adds the following functions to FileMaker Pro:

function name	short description
Serial_Version	use this function to see which version of the plug-in is loaded; this function is also used to register the plug-in
Serial_VersionAutoUpdate	standard version number for AutoUpdate of FileMaker Server
Serial_GetPortNames	returns the names of all serial ports that are available on the computer
Serial_Open	opens a serial port
Serial_Close	closes a serial port
Serial_Receive	receives data from a serial port
Serial_Send	send data to a serial port
Serial_SetDispatchScript	tell the plug-in which script to call when data is received
Serial_DataWasReceived	returns portname if data was received on an open port
Serial_AsciiValueToText	converts (one or more) ASCII values to the equivalent text
Serial_TextToAsciiValue	converts a text string to a list of ASCII values
Serial_Control	suspends and resumes input from a serial port
Serial_Debug	troubleshoot the serial port and test scripting

.

Steps for working with Troi Serial Plug-in

Below you find an overview of the main steps needed to communicate with a serial port:

1 - Find available ports

Use the function “Serial_GetPortNames” to get the names of all serial ports that are available on the computer and let the end user choose a port.

2 - Open the selected port

Use the function “Serial_Open” to open a port. Optionally use the function “Serial_SetDispatchScript” to specify which script is triggered when data comes in from the serial port.

3 - Communicate with the serial port

Use the functions “Serial_Send” and “Serial_Receive” to send and receive data to and from a serial port
You can use other functions, like “Serial_DataWasReceived”, to help you get the data into a FileMaker database.

4 - Close the serial port

At the end of the communication you need to close the serial port.

Specifying the port settings

Default port settings

A serial port can be configured in a lot of ways. These settings can be set by specifying the settings parameter of Serial_Open. If you don't specify any settings the port is initialized to the following settings: a speed of 9600 baud, no parity, 8 data bits, 1 stop bit, no handshaking. If you want to use this setting open the port like this:

```
Set Field[gErrorCode, Serial_Open("-Unused" ; "COM2") ]
```

Specifying other port settings

It is recommended that you set the port settings explicitly. Give the settings by concatenating the desired settings keywords. You specify them like this:

```
Set Field[gErrorCode, Serial_Open("-Unused" ;
           "COM2" ; "baud=9600 parity=none data=8 stop=10 flowControl=XOnXOff" ) ]
```

You can set the speed, the parity, the number of data and stopbits, and the handshaking to use. Note that the order of the keywords and case are ignored. All keywords are optional and should be separated by a space or a return.

Specifying the port speed

The port speed indicates how quickly the data is transported over the serial line.
Allowed values for the port speed are:

```
baud=150    baud=1800   baud=7200   baud=28800  baud=115200
baud=300    baud=2400   baud=9600   baud=38400  baud=230400
baud=600    baud=3600   baud=14400  baud=57600
baud=1200   baud=4800   baud=19200
```

NOTE Not all speeds may be supported on all serial ports. Check the documentation of the computer and the equipment you want to connect.

You need to specify the same speed that the other equipment is using. Higher port speeds can result in loss of data if the serial cable can't cope with this speed. If this happens try a lower speed.

Specifying the bit format options

Data over a serial port is sent in small packets of 4 to 10 bits. These packets consist of 4-8 data bits, followed by a parity bit and stopbits.

Data bits

You can specify the number of the data bits by adding one of the data size keywords to the switch parameter. The most used value is 8 data bits. Allowed values for the number of data bits are:

```
      data=4      data=5      data=6      data=7      data=8
```

Parity bits

You can specify the parity bit by adding one of the following keywords to the switch parameter:

```
      parity=none  parity=odd  parity=even
```

Stop bits

You can specify the number of stopbits by adding one of the following keywords to the switch parameter:

```
      stop=10  stop=15  stop=20
```

Here stop=10 means 1 stop bit, stop=15 means 1.5 stopbit and stop=20 means 2 stopbits.

Specifying the handshaking options

Handshaking is a way to ensure that the transfer of data can be stopped temporarily. This is also called (data) flow control. A serial port can use hardware handshaking and software handshaking. For hardware handshaking to work the serial cable must have wires to support it.

Using the Serial_Open function this plug-in allows a basic way to set the handshaking and also an advanced way, which gives more options, but most users probably don't need.

Basic handshaking options

Basic handshaking has 3 keywords:

```
flowControl=DTRDSR      flowControl=RTSCTS      flowControl=XOnXOff
```

You can specify one or more of these flow control keywords. You should specify at least one of these keywords. Try `flowControl=DTRDSR` as this is mostly supported. `flowControl=DTRDSR` and `flowControl=RTSCTS` are hardware handshaking options, for which you need proper cabling. `flowControl=XOnXOff` is a software based handshake option.

`flowControl=DTRDSR` means that the signal DTR is used for input flow control and DSR for output flow control. `flowControl=RTSCTS` means that the signal RTS is used for input flow control and CTS for output flow control. `flowControl=XOnXOff` uses a XOff character (control-S) and a XOn character (control-Q) to stop input and output flow.

IMPORTANT Do not use `FlowControl=XOnXOff` if you want to transfer binary data, like pictures. This protocol uses two ASCII characters that might also be in the binary data. `FlowControl=XOnXOff` works fine with normal text.

Example 1

```
Set Field[gErrorCode, Serial_Open("-Unused" ;
      "COM2" ; "baud=9600 parity=none data=8 stop=10 flowControl=DTRDSR") ]
```

This will set the port to use DTR/DSR hardware handshaking.

Example 2

```
Set Field[gErrorCode, Serial_Open("-Unused" ; "COM2" ;
     "baud=9600 parity=none data=8 stop=10 flowControl=DTRDSR flowControl=RTSCTS
      flowControl=XOnXOff" ) ]
```

This will set the port to use all 3 types of handshaking in parallel.

Advanced handshaking options

Advanced handshaking options allows you more control over the serial port settings. It enables you to set the handshaking of the output and input separately.

With advanced handshaking you can use the following keywords:

keyword	meaning
inputControl=XOnXOff	use XOnXOff for input handshaking
outputControl=XOnXOff	use XOnXOff for output handshaking
inputControl=RTS	use RTS for input handshaking
outputControl=CTS	use CTS for output handshaking
inputControl=DTR	use DTR for input handshaking
outputControl=DSR	use DSR for output handshaking
DTR=enabled	set DTR signal permanent to high
DTR=disabled	set DTR signal permanent to low
RTS=enabled	set RTS signal permanent to high
RTS=disabled	set RTS signal permanent to low

Below you find how the basic handshaking keywords relate to the advanced handshaking keywords:

basic keyword	=	the same as 2 advanced keywords
flowControl=XOnXOff	=	inputControl=XOnXOff outputControl=XOnXOff
flowControl=RTSCTS	=	inputControl=RTS outputControl=CTS
flowControl=DTRDSR	=	inputControl=DTR outputControl=DSR

The other advanced keywords don't have an equivalent.

NOTE You can mix the basic handshaking keywords with the advanced handshaking keywords, as long as this is sensible.

Example 1

If you want to use DTR handshaking for input flow control and CTS for output flow control use the following settings to open COM1:

```
Set Field[gErrorCode, Serial_Open("-Unused" ; "COM1" ;
  "baud=9600 parity=none data=8 stop=10 outputControl=CTS inputControl=DTR") ]
```

Example 2

If you want to enable the DTR signal and use XOnXOff input flow control use the following settings to open COM1:

```
Set Field[gErrorCode, Serial_Open("-Unused" ; "COM1" ;
      "baud=9600 parity=none data=8 stop=10 DTR=enabled inputControl=XOnXOff") ]
```

Example 3

```
Set Variable[$ErrorCode, Serial_Open("-Unused" ; "COM2" ;
      "baud=9600 data=7 parity=odd stop=20 flowControl=XOnXOff " &
      "outputControl=CTS inputControl=DTR") ]
```

This shows that XOnXOff is used for input and output flow control and also DTR handshaking for input flow control and CTS for output flow control.

Receiving data via script triggering

The Plug-in API for FileMaker Pro 7 and later has an official way to trigger scripts (or dispatch scripts). It is possible on all platforms to trigger scripts by filename and script name. The 4.5 version of the Serial Plug-in implements this triggering. Other ways of triggering are no longer needed.

Functions to implement Dispatch Scripting

The following external functions help in achieving the receiving of data via the Dispatch Script.

Serial_SetDispatchScript	tell the plug-in which (Dispatch) script to trigger when data is received
Serial_DataWasReceived	returns the name of the port when data was received on an open port

The following function is no longer needed, and is no longer present in Troi Serial Plug-in 4.5:

Serial-RestoreSituation

TIP See the example file Terminal.fmp12 for a working example.

Dispatch Scripting using Script Name

This method will trigger a script when data is received on one of the open ports. Usually you set the dispatch script once after you have opened the serial port(s).

Example "Set Dispatch Script with name"

Below you find a sample Set Dispatch Script:

```
Set Field [gErrorCode,  Serial_SetDispatchScript( "-Unused" ; "" ;
            Get(FileName) ; "Process Data Received") ]
If [Left(gErrorCode, 2 ) = "$$"]
      Beep
      Show Message [An error occurred while setting the dispatch script]
      Halt Script
End If
```

This tells the plug-in to trigger the script Process Data Received whenever incoming data from (one of) the serial port(s) is available. In the script Process Data Received you can retrieve the incoming data, and store it, and do any other processing.

Dispatch Scripting for a specific port

This plug-in can also trigger different scripts for different open ports. This is done with the Serial_Open function. This is how this can be done:

Example Dispatch Script for specific port

Below you find a sample "To Menu" Dispatch Script:

```
Set Field [gErrorCode, Serial_Open( "-Unused" ;
            gPortName1 ; "baud=19200 parity=none" ; Get(FileName) ;
            "Process Data Received for 1st Port" ) ]
If [Left(gErrorCode, 2 ) = "$$"]
      Beep
      Show Message ["An error occurred while opening the port." ]
      Halt Script
End If
```

This script will open the port gPortName1 and will trigger script "Process Data Received for 1st Port" when data comes in on this port. If both triggering with Serial_Open and also with Serial_SetDispatchScript has been specified the trigger script specified with Serial_Open takes precedence.

Example Process Data Received script

Below you find a sample "Process Data Received" script, which gets the data from the plug-in into the field mesReceived.

```
Enter Browse Mode []
Set Field [gTempResultReceived, Serial_Receive("-Unused" ;  gPortName ) ]
Set Field [mesReceived, mesReceived & gTempResultReceived]
```

Example "Set Dispatch Script" script

Below you find a sample "Set Dispatch Script" script:

```
Set Field [gErrorCode,  Serial_SetDispatchScript( "-Unused" ;
                              Get(FileName) & "MyTriggerScript")]
If [Left(gErrorCode, 2 ) = "$$"]
      Beep
      Show Message [An error occurred while setting the dispatch script]
      Halt Script
End If
```

Example Start Receiving script

Below you find a sample "Start Receiving" script:

```
Perform Script [Sub-scripts, "Open Serial Port"]
Perform Script [Sub-scripts, "Set Dispatch Script"]
```

When you want to begin receiving perform the "Start receiving script".

Script triggering on a Match String

The Serial plug-in can look for a special match string that has to arrive at the input buffer before it triggers a script. When you specify the dispatch script, you can add the waitstring parameter.

The script step below will set open a port with a dispatch script Process Data Received, which is only triggered after the string OK is received in the input buffer.

```
Set Field [ gErrorCode, Serial_SetDispatchScript( "-Unused" ;
            Get(FileName) &
            "Process Data Received" &
            "OK")   ]
```

The script step below will set a dispatch script Process Data Received , which is only triggered after a CR (carriage return) character, followed by a LF (linefeed) character is received. These are the ASCII characters 0x0D and 0x0A respectively (see the ASCII Table in Appendix A).

Using the Serial_AsciiValueToText function we set the waitstring like this:

```
Set Field [gErrorCode, Serial_SetDispatchScript( "-Unused" ;
            Get(FileName) &
            "Process Data Received" &
            Serial_AsciiValueToText( "-Unused", "OxOD Ox0A") ]
```

There is no longer a length limitation on the waitstring.

Getting the last match string

It is also possible to get the last string of text that matches the match string. You specify this in the Serial_Receive function.

You need to have this script step:

```
Set Field [gResult, Serial_Receive( "-GetLastMatch" ; "COM1" ;  ]
```

Example

We assume, like the example above, to be waiting for match "`<CR><LF>`" and this data comes in:

```
12345<CR><LF>
434343<CR><LF>
5678<CR><LF>
12
```

If we now run the Serial_Receive script step this data is received in the gResult field:

```
5678<CR><LF>
```

All earlier data is discarded.

Controlling input from the serial port

The function “Serial_Control” controls the serial port. With this function you can suspend or resume the incoming data. This command is very useful for devices that send out continuous data, like an electronic weighing scale.

NOTE The buffer will be emptied when the port is suspended. So when you give the resume command only the data received after this command will be received.

NOTE You can continue to send data to the serial port.

Example 1

```
Set Field[ gResult, Serial_Control( "-Suspend" , "Modem port" ) ]
```

This will suspend the incoming stream of data from the Modem port.

```
Set Field[ gResult, Serial_Control( "-Resume" , "Modem port" ) ]
```

This will resume the previously suspended incoming stream of data from the Modem port.

Example 2

Say you have an electronic weighing scale that sends data to the serial port continuously. The data is in this form:

```
1200 kg net CR LF
1199 kg net CR LF
1200 kg net CR LF
1200 kg net CR LF
etc...
```

You are only interested in this data when you are actually weighing something. So the best way to handle this is to open the serial port and then suspend this port. When you want to measure something you send a resume command, and gather a full line of data, then suspend the port again.

You need to define these fields:

gPortName	global text field, to hold the port name
gErrorCode	global text field, to hold the error code in
weight	number field, to store the weight

When starting up the database you issue these commands in a startup script:

```
Set Field[ gPortName,"COM2" ]
Set Field[ gErrorCode, Serial_Open( "-Unused" ; gPortName ; "baud=19200") ]
If[ gErrorCode = 0 ]
      Set Field[ gErrorCode, Serial_Control( "-Suspend" , gPortName ) ]
Endif
```

This will open the port and then wait till further notice.

When the user of the database presses a button you start this Measure Now script:

```
Set Field [gTempResultReceived, ""]
Set Field [gTempBuffer, ""]
Set Field [gNumber, 10]

Comment [Resume the incoming data...]
Set Field [gErrorCode, Serial_Control("-Resume" ; gPortName )]
If [gErrorCode = 0]
  Loop
      Set Field [gTempResultReceived ; Serial_Receive("-Unused" ;  gPortName )]
      Set Field [gTempBuffer, gTempBuffer & gTempResultReceived ]
      Exit Loop If [PatternCount(gTempBuffer , "¶") >= 2 or gErrorCode <> 0]
      Pause/Resume Script [0:00:01]
      Set Field [gNumber, gNumber - 1]
      If [gNumber = 0]
            Set Field [gErrorCode, -1]
      End If
  End Loop
  Set Field [gNumber, Serial_Control("-Suspend" ; gPortName )]
End If
Perform Script [Sub-scripts, Store Measure Results]
```

The Measure Now script resets the buffers, then resumes the incoming data. Inside the loop the data is received until there are 2 returns in the buffer, which means a complete line was received. The script then suspends the port again and then the script Store Measure Results is called to store the results in a record.

To prevent this looping forever when no data is received we also use a counter, gNumber. It starts at 10 and is lowered every time through the loop. After 10x the script gives up and an error code of -1 is set, to get out of the loop.

Here is the Store Measure Results script:

```
If [gErrorCode = 0 and PatternCount(gTempBuffer , "¶") >= 2]
      New Record/Request
      Comment [Cut off at the end of the line]
      Set Field [gTempBuffer, Left(gTempBuffer,
            Position(gTempBuffer, "¶", Length(gTempBuffer) , -1) - 1)]
      Comment [Copy one line from the end...]
      Set Field [Weight, Middle(gTempBuffer,
            Position(gTempBuffer, "¶", Length(gTempBuffer) , -1) + 1,
            Length(gTempBuffer) )]
Else
      Beep
      Show Message [An error occurred!]
End If

Go to Field []
```

This script will create a new record and find the last line in the buffer, and store it in the field Weight.

Serial_AsciiValueToText

Syntax Serial_AsciiValueToText(switches ; ASCIIvalues { ; separator })

Converts (one or more) numbers to their equivalent ASCII characters.

Parameters

<table>
<tr><td>switches</td><td>these alter the behaviour of the function</td></tr>
<tr><td>ASCIIvalues</td><td>one or more numbers in the range from 0-255, separated by a separator</td></tr>
<tr><td>separator</td><td>(optional) the separator between the values. If you omit this parameter " " and | is used.</td></tr>
</table>

Switches can be empty or one of this:

-Encoding=Native	(default) use Unicode encoding for the higher ASCII's 128-255
-Encoding=ASCII_Mac	use Mac ASCII for the higher ASCII's 128-255 (as used in fmp 6)

Returned result

The converted ASCII text

Special considerations

You can also use hexadecimal notation for the numbers. Use 0x00...0xFF to indicate hexadecimal notation.
The graphic rendition of characters greater than 127 is undefined in the American Standard Code for Information Interchange (ASCII Standard) and varies from font to font and from computer to computer and may look different when printed.
Values higher than 255 are ignored.

Renamed from Serial-ToASCII function of Serial Plug-in 2.5.

Example usage

Set Field [text, Serial_AsciiValueToText ("-Unused" ; "65 65 80 13")]
or
Set Field [text, Serial_AsciiValueToText ("-Unused" ; "65|65|80|13")]

This will both result in the text "AAP<CR>" where <CR> is a Carriage Return character

Example 2

Set Field [text, Serial_AsciiValueToText("-Encoding=ASCII_Mac" ; "0x31-0x32-0x33-0x0D-0x0A" ; "-")]

This will result in the text "123<CR><LF>" where <CR> is a Carriage Return character and <LF> is a Line Feed character.

Serial_Close

Syntax Serial_Close(switches ; portname)

Closes a serial port with the specified name.

Parameters

switches	not used, reserved for future use. Leave blank or put "-Unused"
portname	the name of the port to close

Returned result

The returned result is an error code:

0	(no error) the port was closed
$$-4210	(portDoesnotExistErr) port is not available on this computer
$$-4211	(AllPortsNullErr) no serial ports are available on this computer
$$-108	(memFullErr) ran out of memory

Other errors may be returned.

Special considerations

If the portname parameter is "" all ports are closed.

Example usage

This will close the COM3 port:

```
Set Field[ gErrorCode, Serial_Close( "-Unused" ; "COM3") ]
```

Example 2

This will close all open ports:

```
Set Field[ gErrorCode, Serial_Close( "-Unused" ; "" ) ]
```

Serial_Control

Syntax Serial_Control(switches ; portname)

Controls the serial port with the specified name. The port needs to be opened first (see also Serial_Open).

Parameters

switches	the action that needs to be done.
portname	the name of the port to control

Switches can be either:

-Suspend	This will suspend reading the incoming stream of data.
-Resume	This will resume reading the incoming stream of data.

Returned result

The returned result is an error code. An error always starts with 2 dollars, followed by the error code. You should always check for errors when sending by testing if the first two characters are dollars. Returned error codes can be:

0	no error	the data was sent
$$-28	notOpenErr	the port is not open
$$-50	paramErr	there was an error with the parameter

Other errors may be returned.

Special considerations

The buffer will be emptied when the port is suspended. So when you resume, only the data received after you resume will be available. While suspended you can still send data to the serial port.

This function is very useful for devices that send out continuous data, like an electronic weighing scale.

Example usage

```
Set Field[ gResult,  Serial_Control( "-Suspend" ;  "COM1") ]
```

This will suspend the incoming stream of data.

Example 2

For devices that send out continuous data, like an electronic weighing scale, you open the port and suspend the incoming data. Then when you want a reading you resume the incoming stream. The script will be like this:

```
Set Field[gErrorCode,  Serial_Open( "-Unused" ;  gPortName ) ]
Set Field[gErrorCode,  Serial_Control( "-Suspend" ;  gPortName ) ]
# do other stuff here, until you need data from the device ...
Set Field[gErrorCode,  Serial_Control( "-Resume" ;  gPortName ) ]
# read data until you got the expected data:
Set Field[gErrorCode,  Serial_Read( "-Unused" ;  gPortName ) ]
# This will suspend reception of data from the port in field gPortName:
Set Field[gErrorCode,  Serial_Control( "-Suspend" ;  gPortName ) ]
```

Serial_DataWasReceived

Syntax Serial_DataWasReceived(switches)

Returns the name of the port when data was received on a serial port. Use this function to see if this is an event that needs to be handled.

Parameters

switches	determines what is returned

Switches currently needs to be:

-FirstPortname	return the name of the first port for which data was received

Returned result

The returned result is either:

""	no data received
"portname"	data was received in the buffer for this port

Special considerations

When this function returns something else than "" you can get the data with the function Serial_Receive.

If you have opened more than one port the plug-in just indicates the first portname (in its internal list) that has received data.
If you want to read the data from *all* open ports you should read this in a loop: read the data from the first portname and then loop and read from the next portname until there is no more data.

In some cases you might want to read the data from the ports in your own order. That is fine with the plug-in. The Serial_DataWasReceived function just helps in determining which port has data.

Example usage

```
Set Field [ gPortname , Serial_DataWasReceived( " -FirstPortname " ) ]
If[ gPortname = "COM1"]
        Perform Script [Sub-scripts, “Process Data Received COM1"]
Else
        ... do something else
End If
```

Serial_Debug

Syntax Serial_Debug(switches)

Use this function to troubleshoot the serial port and test scripting.

Parameters

switches	determine the behaviour of the function

switches can be one of this:

-BeepWhenDataArrives	the plug-in will beep when data arrives at the serial port
-BeepOff	the plug-in no longer beeps when data arrives at the serial port
-TestTriggerScript	the plug-in will trigger the dispatch script with some test data

You can also add this switch in combination with the -TestTriggerScript switch:

-UseTextAllBytes0To255	sends a string with all bytes from 0-255 as the test data to the trigger script.

Returned result

If successful it returns 0. If unsuccessful it returns an error code starting with $$ and the error code. Possible error codes are:

$$-28	notOpenErr	make sure you have opened a port and specified a dispatch script
$$-50	paramErr	there was an error with the parameter (an unknown switch was given)

Other errors may be returned.

Special considerations

To be able to test the trigger script the serial port must be opened and a dispatch script must be specified.

See the Debug.fmp12 example file.

NOTE The wait time to trigger the script is now 5 seconds (was 10 seconds).

Example usage

Set Field [gErrorCode, Serial_Debug("-BeepWhenDataArrives")]

The plug-in will now beep when data arrives at the serial port. If data keeps on coming in, the plug-in will beep every 2 seconds.

How to test if data comes in:
- Set the plug-in to beep if data comes in.
- Get the device attached to the serial port to send data to the serial port.
- When you hear beeps you know data is arriving at the serial port and in the plug-in.
- To stop the beeping use this command:

Set Field [gErrorCode, Serial_Debug("-BeepOff")]

Example 2

Serial_Debug("-TestTriggerScript") will trigger the dispatch script after 5 seconds.

You can also give this command:
Serial_Debug("-TestTriggerScript -UseTextAllBytes0To255")
This will trigger the dispatch script after 5 seconds, the data received is a string with all the bytes from 0 to 255.

Serial_GetPortNames

Syntax Serial_GetPortNames(switches)

Returns the names of all serial ports that are available on the computer.

Parameters

switches	not used, reserved for future use. Leave blank or put "-Unused"

Returned result

The returned result is a list of serial ports that are available on the computer that is running FileMaker Pro. Each available port is on a different line. On Mac OS X a result can be for example:

Internal Modem
Bluetooth-Modem

On Windows the result will be for example:

COM2
COM4

Use this function to let the user of the database choose which port to open. Store the name of the chosen port in a global field. You can then check the next time the database is opened whether the portname is still present and ask the user if he wants to change his preference.

If an error occurs an error code is returned. Returned error codes can be:

$$-108	memFullErr	Ran out of memory

Other errors may be returned.

Special considerations

Starting with version 2.9 the plug-in will detect all existing serial ports on Windows, instead of always returning with COM1...COM4.

Example usage

Set Field [result, Serial_GetPortNames("")]

This returns the names of the serial ports available. On our Intel Mac with a Keyspan USB to serial adaptor installed, the result is this list:

KeySerial1
USA28X1d1P1.1
USA28X1d1P2.2
Bluetooth-PDA-Sync
BlueSerialPort-2
Bluetooth-Modem

The first 3 ports are supplied by the Keyspan adaptor. The last 3 ports are serial ports over a Bluetooth wireless connection. To be able to use these you need proper Bluetooth hardware.

On windows the names of the ports might be:

COM2
COM3
COM4

Serial_Open

Syntax Serial_Open(switches ; portname ; settings ; filename ; scriptname)

Opens a serial port with this name and the specified parameters.

Parameters

switches	(optional) specifies how this function receives the data
portname	the name of the port to open
settings	(optional) specifies the setting of the port like the speed of the port, parity, etc.
filename	(optional) the name of the file which contains the script to trigger when data comes in
scriptname	(optional) specifies the name of the script to trigger when data comes in

switches can be one of this:
-ResumeWhenScriptPaused when the trigger script needs to run, and an other script is already paused, the paused script will resume after the triggerscript is finished.
-NoIdleWaitTime don't add idle wait times, the triggering will be faster, but the plug-in will need more computer time.

Returned result

Returned result is an error code:

0	no error
$$-50	(paramErr) there was an error with the parameter
$$-108	(memFullErr) ran out of memory
$$-97	(portInUse) could not open port, the port is in use
$$-4210	(portDoesnotExistErr) port with this name is not available on this computer
$$-4211	(allPortsNullErr) no serial ports are available on this computer

Other errors may be returned.

Special considerations

If you specify a filename and scriptname any scripts specified with the function "Serial_SetDispatchScript" will be ignored for this port.

If you specify a filename you must also provide a scriptname.

Windows only: if you have more than 4 ports, you might get error $$-4210. In this case call Serial_GetPortNames("-portCount=8") first. You can also use a different count.

Example usage

```
Set Field[gErrorCode,       Serial_Open( "-Unused" ; "COM2; "baud=19200 parity=none
            data=8 stop=10 flowControl=DTRDSR  flowControl=RTSCTS") ]
```

will open the COM2 port with a speed of 19200 baud and the specified options.

Example 2

```
Set Field[gErrorCode,
            Serial_Open( "-Unused" ; gPortName1 ;
                        gSpeed & " " & gStopBits & " " & gDataBits & " " & gParity &  " "& gFlowControl ;
                        Get(FileName) ;
                        "Process Data Received for 1st Port"
                        )
        ]
```

Serial_Open

This will open the port in field gPortName1with the specified speed and other options. When data comes in the script "Process Data Received for 1st Port" in the current filename will be triggered.

Serial_Receive

Syntax Serial_Receive(switches ; portname)

Receives data from a serial port with the specified name. The port needs to be opened first (See Serial_Open). If no data is available an empty string is returned: "".

Parameters

switches	(optional) specifies how this function receives the data
portname	the name of the port to receive data from

switches can be left empty or can be:

-GetLastMatch	get the last string of text that matches the match string

You can also add one of these encoding switches, which determine how the incoming bytes are interpreted.

-Encoding=ASCII_DOS	
-Encoding=ASCII_Windows	(Windows ANSI)
-Encoding=ASCII_Mac	(Mac Roman)
-Encoding=ISO_8859_1	(Windows Latin-1)
-Encoding=UTF8	
-Encoding=BytesOnly	returns all received bytes as the same Unicode byte values (Unicode 0 to 255)

-ConvertBytesToNumbers	converts all received bytes to their numeric value, with space as separator

Returned result

The returned result is the data received or an error code. An error always starts with 2 dollars, followed by the error code. You should always check for errors when receiving, by testing if the first two characters are dollars.

Returned error codes can be:

$$-28	notOpenErr	The port is not open
$$-108	memFullErr	Ran out of memory
$$-50	paramErr	There was an error with the parameter
$$-4210	portDoesnotExistErr	Port with this name is not available on this computer
$$-4211	allPortsNullErr	No serial ports are available on this computer
$$-207	notEnoughBufferSpace	The input buffer is full

Other errors may be returned.

Special considerations

The plug-in will get any data that is received at the time the function is called. This might not be all data coming in. You might need to wait and append new data coming in at a later time.

When you use the -GetLastMatch switch the last matching string of text is returned. Older text is discarded.

Please be aware that only the ASCII characters 0...255 will be received, as a serial port uses 8 bit characters.

When using the new switch -ConvertBytesToNumbers all received bytes are converted to their numeric value. It will for example return "65 66 67 ", when receiving the bytes "ABC". Note that each number is followed by a space, as separator, including after the last number.

Example usage

Set Field[gResult, Serial_Receive("-Unused" ; "SerialPort1")]

This will receive data from the SerialPort1. It might return "All the world is a sta". If you call it again later new data may have come in and the result might be "ge and we are merely players." It is best to concatenate the data coming in.

Example 2

Below you find a "Receive Data" script for receiving data into a global text field gTempResultReceived. The script tests for errors.

We assume that in your FileMaker file the following fields are defined:

gPortName	Global, text, contains the name of the previously opened port
gTempResultReceived	Global, text
gTotalResult	Global, text, can also be a normal text field

Add the following script steps:

```
Set Field [gTempResultReceived, Serial_Receive("-Unused" ;  gPortName) ]
If [Left(gTempResultReceived, 2 ) = "$$"]
        Beep
        If [gTempResultReceived = "$$-28"]
                Show Message [Open the port first]
        Else
                If [gTempResultReceived = "$$-207"]
                        Show Message [Buffer overflow error.]
                Else
                        Show Message [An error occurred!]
                End If
        End If
        Halt Script
Else
        # no error, so concatenate the data somewhere and do your stuff.
        Set Field [gTotalResult , gTotalResult & gTempResultReceived ]
     #  .... add your own steps here ...
End If
```

Serial_Reinitialize

Syntax Serial_Reinitialize(switches)

Tell the plug-in to re-initialize itself and look which serial ports are available on the system now.

Parameters

switches	not used, reserved for future use. Leave blank or put "-Unused"

Returned result

If successful it returns 0. If unsuccessful it returns an error code starting with $$ and the error code. Possible error codes are:

$$-4211	kErrAllPortNull	no serial ports found

Other errors may be returned in the future.

Special considerations

This function will close all ports first. Then the plug-in looks again for available ports.

You can use this function when new or different serial ports are added dynamically, for example if you plug-in a USB To Serial adapter.

Example usage

Set Field [gErrorCode, Serial_Reinitialize("")]

Serial_Send

Syntax Serial_Send(switches ; portname ; data)

Sends data to the serial port with the specified name. The port needs to be opened first (See also Serial_Open).

Parameters

switches	not used, reserved for future use. Leave blank or put "-Unused"
portname	the name of the port to send data to
data	the text data that is to be sent to the serial port

Returned result

The returned result is an error code. An error always starts with 2 dollars, followed by the error code. You should always check for errors when sending, by testing if the first two characters are dollars.

Returned error codes can be:

0	no error	the data was sent
$$-28	notOpenErr	The port is not open
$$-108	memFullErr	Ran out of memory
$$-50	paramErr	There was an error with the parameter
$$-4210	portDoesnotExist	A port with this name is not available on this computer
$$-4211	allPortsNullErr	No serial ports are available on this computer
$$-207	notEnoughSpace	The output buffer is full

Other errors may be returned.

Special considerations

Make sure you use a text field for the data. Other field types, like containers are currently not supported.

Please be aware that only the ASCII characters 0...255 will be transmitted, as a serial port wants 8 bit characters.

Example usage

```
Serial_Send( "-Unused" ; "Modem port" ;  "So long") ]
```

This will send the string " So long" to the Modem port.

```
Set Field[ gResult, Serial_Send( "-Unused" ;  gPortName ; textToSend ) ]
```

This will send the text in the field textToSend to the port in the field gPortName.

Example 2

Below you find a "Send Data" script for sending data from a global text field gTextToSend. The script tests for errors.

We assume that in your FileMaker file the following fields are defined:

gPortName	Global, text, contains the name of the previously opened port
gTextToSend	Global, text, can also be a normal text field
gErrorCode	Global, text

Add the following script steps:

```
Set Field [gErrorCode, Serial_Send( "-Unused" ; gPortName ; gTextToSend) ]
```

Serial_Send

```
If [Left(gErrorCode, 2 ) = "$$"]
    Beep
    If [gErrorCode = "$$-28"]
        Show Message [Open the port first]
    Else
        If [gErrorCode = "$$-207"]
            Show Message [Buffer overflow error.]
        Else
            Show Message [An error occurred while sending!]
        End If
    End If
    Halt Script
End If
```

Serial_SetDispatchScript

Syntax Serial_SetDispatchScript(switches ; portname ; filename ; scriptname ; waitstring)

Sets the script to trigger when data is received. If you give an empty filename parameter "", the dispatch script is removed.

Parameters

switches modifies how this function behaves
portname (optional) the name of the serial port to be coupled to this trigger script. If you leave this empty the trigger script applies to all ports.
filename the name of the file with the Dispatch Script
scriptname the name of the script to be triggered
waitstring (optional) wait for a string of characters before triggering a script.

switches can be one of this:
-ResumeWhenScriptPaused when a trigger script needs to run, and an other script is already paused, the paused script will resume after the triggerscript is finished.

Returned result

The returned result is an error code. An error always starts with 2 dollars, followed by the error code. You should always check for errors. Returned error codes can be:

0	no error	the Dispatch Script was set
$$-50	paramErr	There was an error with the parameter

Other errors may be returned.

Special considerations

See also the User Manual under Dispatch Scripting for more details.
If the filename parameter is empty, the dispatch script is removed and the plug-in will no longer trigger. Note that this will only remove the general port trigger.

Example usage

```
Set Field[ gErrorCode, Serial_SetDispatchScript(   "-Unused" ; "" ;
          Get(FileName) ; "Read Script"  ; "OK") ]
```

This will set the Dispatch Script for all ports to the script "Read Script" of the current file. The script will not be triggered before the string "OK" is found.

```
Set Field[ gErrorCode, Serial_SetDispatchScript(    "-Unused" ;  "COM2" ; Get(FileName) ;  "TriggerScriptCOM2"  ;  ]
```

This will set the Dispatch Script for the COM2 port to the script "TriggerScriptCOM2" of the current file.

Example 2

```
Set Field[ gErrorCode, Serial_SetDispatchScript(  "-Unused" ; "" ; "" ) ]
```

This will reset all the dispatch scripts. Although the incoming data is buffered, no action is taken when data is received. You can still get the data out by calling the Serial_Receive() function.

Serial_TextToAsciiValue

Syntax Serial_TextToAsciiValue(switches ; text { ; separator })

Converts text to one or more ASCII values.

Parameters

switches	these alter the behaviour of the function
text	the text to convert
separator	(optional) the separator between the values, if you omit this parameter " " is used.

Switches can be empty or one of this:

-Encoding=Native	(default) use Unicode encoding for the higher ASCII's 128-255
-Encoding=ASCII_DOS	use OEM DOS ASCII for the higher ASCII's 128-255
-Encoding=ASCII_Windows	use Ansi Windows ASCII for the higher ASCII's 128-255
-Encoding=ASCII_Mac	use Mac ASCII for the higher ASCII's 128-255 (as used in fmp 6)

Returned result

one or more ASCIIvalues (in the range from 0-255) separated by spaces
If a character is out of range, a ? (question mark) is returned on the place of the character

Special considerations

The graphic rendition of characters greater than 127 is undefined in the American Standard Code for Information Interchange (ASCII Standard) and varies from font to font and from computer to computer and may look different when printed.

Example usage

Say you have a text "AAP<CR>", where <CR> is a Carriage Return character. Then call the function like this:

```
Serial_TextToAsciiValue ("-unused" ; "AAP<CR>")
```

This will result in "65 65 80 13"

Example 2

Set Field [text, Serial_TextToAsciiValue ("-unused" ; "AAP<CR>"; ",")] , where <CR> is a Carriage Return character.
This will result in "65,65,80,13".

Serial_Version

Syntax Serial_Version(switches)

Use this function to see which version of the plug-in is loaded.
Note: This function is also used to register the plug-in.

Parameters

switches	determine the behaviour of the function

switches can be one of this:

-GetVersionString	the version string is returned (default)
-GetVersionNumber	returns the version number of the plug-in
-ShowFlashDialog	shows the Flash Dialog of the plug-in (returns 0)
-GetRegistrationState	get the registration state of the plug-in: 0 = not registered ; 1 = registered
-UnregisterPlugin	sets the registration state of the plug-in to unregistered

If you leave the parameter empty the version string is returned.

Returned result

The function returns ? if this plug-in is not loaded. If the plug-in is loaded the result depends on the input parameter. It is either a:

-GetVersionString:
If you asked for the version string it will return for example "Serial Plug-in 3.0"

-GetVersionNumber:
If you asked for the version number it returns the version number of the plug-in x1000. For example version 3.0 will return number 3000.

-ShowFlashDialog:
This will show the flash dialog and then return the error code 0.

Special considerations

Important: always use this function to determine if the plug-in is loaded. If the plug-in is not loaded use of external functions may result in data loss, as FileMaker will return a question mark from any external function that is not loaded.

Example usage

Serial_Version("") will for example return "Serial Plug-in 3.0"

Example 2

Serial_Version("-GetVersionNumber") will return 3000 for version 3.0
Serial_Version("-GetVersionNumber") will return 4510 for a possible future version 4.5.1

So for example to use a feature introduced with version 3.0 test if the result is equal or greater than 3000.

Serial_VersionAutoUpdate

Syntax Serial_VersionAutoUpdate

Use this function to see which version of the plug-in is loaded, formatted for FileMaker Server's AutoUpdate function. Returns 8 digit number to represent an AutoUpdate version.

Parameters

none

Returned result

The function returns ? if this plug-in is not loaded. If the plug-in is loaded the result is a version number, it is returned in the format aabbccdd where every letter represents a digit of the level, so versions can be easily compared.

Special considerations

The Serial_VersionAutoUpdate function is part of an emerging standard for FileMaker plug-ins of third party vendors of plug-ins. The version number can be easily compared, when using the Autoupdate functionality of FileMaker Server.

Example usage

Serial_VersionAutoUpdate will return 03010000 for version 3.1
Serial_VersionAutoUpdate will return 03060203 for version 3.6.2.3

So for example to use a feature introduced with version 3.1 test if the result is equal or greater than 03010000.

Appendix A: ASCII Table

Char	Dec	Hex	Control	Description
NUL	0	0x00	^@	null (end of C string)
SOH	1	0x01	^A	start of heading
STX	2	0x02	^B	start of text
ETX	3	0x03	^C	end of text
EOT	4	0x04	^D	end of transmission
ENQ	5	0x05	^E	enquiry
ACK	6	0x06	^F	acknowledge
BEL	7	0x07	^G	bell
BS	8	0x08	^H	backspace
TAB	9	0x09	^I	horizontal tab
LF	10	0x0A	^J	line feed
VT	11	0x0B	^K	vertical tab
FF	12	0x0C	^L	form feed
CR	13	0x0D	^M	carriage return
SO	14	0x0E	^N	shift out
SI	15	0x0F	^O	shift in
DLE	16	0x10	^P	data line escape
DC1	17	0x11	^Q	device control 1 (X-ON)
DC2	18	0x12	^R	device control 2
DC3	19	0x13	^S	device control 3 (X-OFF)
DC4	20	0x14	^T	device control 4
NAK	21	0x15	^U	negative acknowledge
SYN	22	0x16	^V	synchronous idle
ETB	23	0x17	^W	end transmission block
CAN	24	0x18	^X	cancel
EM	25	0x19	^Y	end of medium
SUB	26	0x1A		substitute
ESC	27	0x1B	^[	escape
FS	28	0x1C	^\	file separator
GS	29	0x1D	^]	group separator
RS	30	0x1E	^^	record separator
US	31	0x1F	^_	unit separator

Char	Dec	Hex	Description
sp	32	0x20	space
!	33	0x21	
"	34	0x22	
#	35	0x23	
$	36	0x24	
%	37	0x25	
&	38	0x26	
'	39	0x27	
(	40	0x28	
)	41	0x29	
*	42	0x2A	
+	43	0x2B	
,	44	0x2C	
-	45	0x2D	
.	46	0x2E	
/	47	0x2F	
0	48	0x30	
1	49	0x31	
2	50	0x32	
3	51	0x33	
4	52	0x34	
5	53	0x35	
6	54	0x36	
7	55	0x37	
8	56	0x38	
9	57	0x39	
:	58	0x3A	
;	59	0x3B	
<	60	0x3C	
=	61	0x3D	
>	62	0x3E	
?	63	0x3F	
@	64	0x40	

Char	Dec	Hex
A	65	0x41
B	66	0x42
C	67	0x43
D	68	0x44
E	69	0x45
F	70	0x46
G	71	0x47
H	72	0x48
I	73	0x49
J	74	0x4A
K	75	0x4B
L	76	0x4C
M	77	0x4D
N	78	0x4E
O	79	0x4F
P	80	0x50
Q	81	0x51
R	82	0x52
S	83	0x53
T	84	0x54
U	85	0x55
V	86	0x56
W	87	0x57
X	88	0x58
Y	89	0x59
Z	90	0x5A
[	91	0x5B
\	92	0x5C
]	93	0x5D
^	94	0x5E
_	95	0x5F
`	96	0x60

Appendix A: ASCII Table (continued)

Char	Dec	Hex
a	97	0x61
b	98	0x62
c	99	0x63
d	100	0x64
e	101	0x65
f	102	0x66
g	103	0x67
h	104	0x68
i	105	0x69
j	106	0x6A
k	107	0x6B
l	108	0x6C
m	109	0x6D
n	110	0x6E
o	111	0x6F
p	112	0x70
q	113	0x71
r	114	0x72
s	115	0x73
t	116	0x74
u	117	0x75
v	118	0x76
w	119	0x77
x	120	0x78
y	121	0x79
z	122	0x7A
{	123	0x7B
|	124	0x7C
}	125	0x7D
~	126	0x7E
Del	127	0x7F
Ä	128	0x80
Å	129	0x81
Ç	130	0x82
É	131	0x83
—	132	0x84
Ö	133	0x85
Ü	134	0x86
·	135	0x87
à	136	0x88
â	137	0x89
ä	138	0x8A
ã	139	0x8B
å	140	0x8C
ç	141	0x8D
é	142	0x8E
è	143	0x8F
ê	144	0x90
ë	145	0x91
í	146	0x92
ì	147	0x93
î	148	0x94
ï	149	0x95
ñ	150	0x96
ó	151	0x97
ò	152	0x98
ô	153	0x99
ö	154	0x9A
õ	155	0x9B
ú	156	0x9C
ù	157	0x9D
û	158	0x9E
ü	159	0x9F
†	160	0xA0

Char	Dec	Hex
°	161	0xA1
¢	162	0xA2
£	163	0xA3
§	164	0xA4
•	165	0xA5
¶	166	0xA6
ß	167	0xA7
®	168	0xA8
©	169	0xA9
™	170	0xAA
´	171	0xAB
¨	172	0xAC
≠	173	0xAD
Æ	174	0xAE
Ø	175	0xAF
∞	176	0xB0
±	177	0xB1
≤	178	0xB2
≥	179	0xB3
¥	180	0xB4
µ	181	0xB5
∂	182	0xB6
∑	183	0xB7
∏	184	0xB8
π	185	0xB9
∫	186	0xBA
ª	187	0xBB
º	188	0xBC
Ω	189	0xBD
æ	190	0xBE
ø	191	0xBF
¿	192	0xC0
¡	193	0xC1
¬	194	0xC2
√	195	0xC3
ƒ	196	0xC4
≈	197	0xC5
∆	198	0xC6
«	199	0xC7
»	200	0xC8
…	201	0xC9
	202	0xCA
À	203	0xCB
Ã	204	0xCC
Õ	205	0xCD
Œ	206	0xCE
œ	207	0xCF
–	208	0xD0
—	209	0xD1
“	210	0xD2
”	211	0xD3
‘	212	0xD4
’	213	0xD5
÷	214	0xD6
◊	215	0xD7
ÿ	216	0xD8
Ÿ	217	0xD9
⁄	218	0xDA
€	219	0xDB
‹	220	0xDC
›	221	0xDD
ﬁ	222	0xDE
ﬂ	223	0xDF
‡	224	0xE0

Char	Dec	Hex
·	225	0xE1
‚	226	0xE2
„	227	0xE3
‰	228	0xE4
Â	229	0xE5
Ê	230	0xE6
Á	231	0xE7
Ë	232	0xE8
È	233	0xE9
Í	234	0xEA
Î	235	0xEB
Ï	236	0xEC
Ì	237	0xED
Ó	238	0xEE
Ô	239	0xEF
	240	0xF0
Ò	241	0xF1
Ú	242	0xF2
Û	243	0xF3
Ù	244	0xF4
ı	245	0xF5
ˆ	246	0xF6
˜	247	0xF7
¯	248	0xF8
˘	249	0xF9
˙	250	0xFA
˚	251	0xFB
	252	0xFC
˝	253	0xFD
	254	0xFE
ˇ	255	0xFF