Troi Serial Plug-in 2.2 USER GUIDE

October 2001

Troi Automatisering
Vuurlaan 18
2408 NB Alphen a/d Rijn
The Netherlands
Fax: +31-172-470539

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

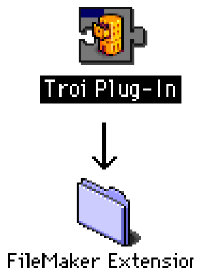
Table of Contents

Click on a line to jump to that subject

Installing plug-ins

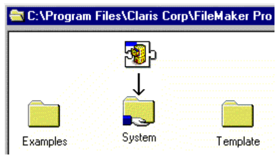
For MacOS:

- Quit FileMaker Pro.
- Put the file “Troi Serial Plug-in” and the file “Troi Serial X extension” from the folder “MacOS Plug-in” into the “FileMaker Extensions” folder in the FileMaker Pro folder.
- If you have installed previous versions of this plug-in, you are asked: “An older item named “Troi Serial Plug-in” 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 the Troi Serial Plug-in is used it will display a dialog box, indicating that it is loading and showing the registration status.



For Windows:

- Quit FileMaker Pro.
- Put the file "trserial.fmx" from the directory "Windows Plug-in" into the "SYSTEM" subdirectory in the FileMaker Pro directory.
- If you have installed previous versions of this plug-in, you are asked: “This folder already contains a file called 'trserial.fmx'. Would you like to replace the existing file with this one?’. Press the Yes button.
- Start FileMaker Pro. The Troi Serial Plug-in will display a dialog box, indicating that it is loading 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.fp5" to see how to use the plug-in's functions. There is also a Function overview available.

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. 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 mistake 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 our 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!

What can this plug-in do?

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

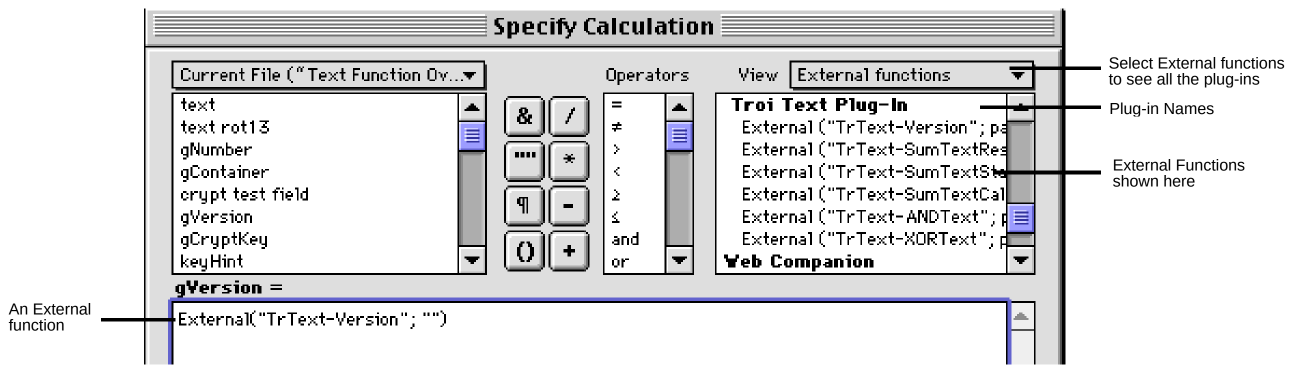
NOTE USB ports are not supported. USB is a bus protocol that can be used from various purposes and devices, like keyboards, harddisks, CD-ROM drives, adaptors, cameras. All these devices need specific drivers. We have currently no plans to create a USB plug-in. Note however that the Troi Serial Plug-in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

Getting started

Using external functions

The 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:

You use special syntax with external functions: External("function name",parameter) where function name is the name of an external function. The parameter is required, even if it's only "". 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 ScriptMaker Script.



```
External("TrText-Version"; pa
External("TrText-SumTextRes
External("TrText-SumTextSt
External("TrText-SumTextCal
External("TrText-ANDText"; p
External("TrText-XORText"; p
```

```
gVersion =
External("TrText-Version"; "")
```

IMPORTANT In the United States, commas act as list separators in functions. In other countries semicolons might be used as list separators. The separator being used depends on the operating system your computer uses, as well as the separator used when the file was created. All examples show the functions with commas. For example: `External("Serial-Version", "")` will become `External("Serial-Version"; "")` in such a file.

Where to add the External Functions?

External functions for this plug-in are intended to be used in a 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.

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 ScriptMaker Script called "Get Serial Port Names". Delete all steps and then add the following script step:

```
Set Field [ gPortNames, External("Serial-GetPortNames", "")]
```

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, like `Serial-GetPortNames` are case sensitive. Be sure to spell them right, 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!

Summary of functions

The Troi Serial Plug-in adds the following functions:

function name	short description
Serial-Version	check for correct version of the plug-in
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 if data was received on a open port
Serial-RestoreSituation	tell the plug-in to bring the original file back to the front
Serial-ToASCII	converts (one or more) numbers to their equivalent ASCII characters
Serial-Control	suspends and resumes input from a serial port

Steps for working with the 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” and “Serial-RestoreSituation” 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 switches. If you don't specify any switches the port is initialised 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, External("Serial-Open", "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, External("Serial-Open",
        "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 quick a 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 packet of 4 to 10 bits. This packet consists 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 datasize 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 giving 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 giving 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 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=RTSCT` 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, External("Serial-Open",
            "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, External("Serial-Open",
            "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 an 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 DSRfor 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 a 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, External("Serial-Open",
              "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, External("Serial-Open",
              "COM1| baud=9600 parity=none data=8 stop=10
                     DTR=enabled inputControl=XOnXOff") ]
```

Example 3

```
Set Field[gErrorCode, External("Serial-Open",
                    "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 Dispatch Scripting™

FileMaker 5.0 added support for ActiveX on Windows. Together with Apple Event support on the Mac it is now possible on all platforms to trigger scripts by name. The 2.0 version of the Serial Plug-in implemented these automation features, by extending the Dispatch Scripting mechanism. It is now possible to tell the plug-in the name of the script to be triggered. It is no longer needed that this script is visible in the Scripts Menu.

NOTE Starting with version 2.2 it is now also possible to trigger scripts via names when running FileMaker 4 on the Windows platform, or runtimes created with FileMaker Developer 5 and FileMaker Developer 5.5 on Windows.
For compatibility the original Dispatch Scripting via a key (see below) is still available.

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 call when data is received
........Serial-DataWasReceived	returns 1 when data was received on a open port
........Serial-RestoreSituation	tell the plug-in to bring the original file back to the front

-> See the sample file **Dispatch.fp5** for a working example.

Dispatch Scripting using Script Name

This method of triggering a script when there is data received is the preferred way. Usually you set the dispatch script once after you have opened the serial port.

Example "Set Dispatch Script with name"

Below you find a sample `Set Dispatch Script`:

```
Set Field [gErrorCode,     External("Serial-SetDispatchScript",
              Status(CurrentFileName) & "| scriptname=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 using a Key

This plug-in also has a different way to execute a script when data has been received. This is done via a Dispatch Script with a key. If you want this functionality you need to implement the Dispatch functions in your database. This is how this can be done:

During development

You have to implement this once:
- write the Dispatch Script or change an existing script
- include the Dispatch Script in the menu, so it can be called from the keyboard with control-1 to control 9 (Windows) or command-1 to command-9 (Mac)
- write a "Start receiving script" that
    • opens the serial port
    • and tells the plug-in which is the Dispatch Script.

When Running the database

When the database is running and you want to begin receiving:
- perform the "Start receiving script".

This tells the plug-in for example that the Dispatch Script can be called from the keyboard with control-1 (Windows) or command-1 (Mac).

This is what happens when data arrives:
- the plug-in will bring the database file to the front and simulate a press on the keyboard:control-1 (Windows) or command-1(Mac).
- this will start the Dispatch Script, which can handle the receiving of the data.

NOTE You can still use the Dispatch Script for other actions, so this doesn't cost a place in the menu. That's why we call it a dispatching script: when called it determines if it was called because there was data received and if yes it will dispatch the processing.

Example Dispatch Script

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

```
If [External("Serial-DataWasReceived", "")]
      Perform Script [Sub-scripts, "Process Data Received"]
Else
      Enter Browse Mode []
      Go to Layout [“Menu”]
      Halt Script
End If
```

This script checks if there is data received. If this is the case it dispatches to the script `"Process Data Received"` which receives the data and puts it into a field. Else it will do its normal business (going to a menu).

Make sure you include this script in the menu. We assume this script can be performed with the keyboard shortcut :control-1 (Windows) or command-1 (Mac)

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 []
Perform Script [Sub-scripts, "Receive Data in global gTempResultReceived"]
Set Field [mesReceived, mesReceived & gTempResultReceived]
Set Field [gErrorCode, External("Serial-RestoreSituation", "") ]
```

Example "Set Dispatch Script" Script

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

```
Set Field [gErrorCode,    External("Serial-SetDispatchScript",
                                 Status(CurrentFileName) & "| scriptkey=1")]
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 the it triggers a script. When you specify the dispatch script, you can add the `waitformatch` parameter.

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

```
Set Field [ gErrorCode, External("Serial-SetDispatchScript" ,
             Status(CurrentFileName) &
             "| scriptname=Process Data Received" &
             "| waitformatch=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 `ToASCII` function we set the matchstring like this:

```
Set Field [gErrorCode, External("Serial-SetDispatchScript",
             Status(CurrentFileName) &
             "| scriptname=" & "Process Data Received" &
             "| waitformatch=" & External("Serial-ToASCII", "OxOD|Ox0A") ]
```

You can specify any string up to 25 characters.

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, External("Serial-Control" , "Modem port|suspend") ]
```

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

```
      Set Field[ gResult, External("Serial-Control" , "Modem port|resume") ]
```

This will resume the previously resumed 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, the suspend the port again.

You need to define these fields:

gPortName	global text field, to hold the portname
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 commando in a **startup script**:

```
      Set Field[ gPortName,"COM2" ]
      Set Field[ gErrorCode, External("Serial-Open" , gPortName & "|baud=19200") ]
      If[ gErrorCode = 0 ]
       Set Field[ gErrorCode, External("Serial-Control" , gPortName & "|suspend") ]
      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, External("Serial-Control", gPortName & "| resume")]
If [gErrorCode = 0]
  Loop
       Perform Script [Sub-scripts,  Receive Data in global gTempResultReceived ]
       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, External("Serial-Control", gPortName & "| suspend")”]
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(gTempBuf
)]
       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`.

Function Reference

Serial-Close

Syntax Set Field[result, External("Serial-Close" , "portname")]

Closes a serial port with the specified name.

Parameters

portname	*the name of the port to close*

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

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.

Example usage

This will close the COM3 port:

```
    Set Field[ gErrorCode, External("Serial-Close" , "COM3") ]
```

Example 2

This will close all open ports:

```
    Set Field[ gErrorCode, External("Serial-Close" , "") ]
```

Serial-Control

Syntax Set Field[result, External("Serial-Control" , "portname | switch")]

Controls the serial port with the specified name . The port needs to be opened first (See also Serial-Open).

Parameters

portname	*the name of the port to control*
switch	*the action that needs to be done. This 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 send
$$-28	notOpenErr	The port is not open
$$-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, External("Serial-Control" , "COM1| suspend")]

This will suspend the incoming stream of data.

Example 2

Set Field[gErrorCode, External(""Serial-Control" , gPortName & "|resume")]

This will resume reception of data from the port in field gPortName.

Serial-DataWasReceived

Syntax SetField[result, External("Serial-DataWasReceived", ""]

Returns 1 when data was received on a serial port. Use this function to see if this is an event that needs to be handled.

Parameters

no parameters leave empty for future use.

Returned result

The returned result is an boolean value. Returned is either:

0	no data received
1	data was received in the buffer

When this function returns 1 you can get the data with the function Serial-Receive.

Example usage

```
If[ External("Serial-DataWasReceived",  "") ]
        Perform Script [Sub-scripts, “Process Data Received”]
Else
        ... do something else
Endif
```

Serial-GetPortNames

Syntax Set Field[result, External("Serial-GetPortNames" , "")]

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

Parameters

no parameters, leave empty for future use.

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 a Mac a typical result will be:

Printer Port¶
Modem Port¶

On Windows the result will be:

COM1¶
COM2¶
COM3¶
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.

Example usage

```
Set Field [ result, External(Serial-GetPortNames, "") ]
```

This might return "Internal Modem Port".

Serial-Open

Syntax Set Field[result, External("Serial-Open" , "portname|switches|filename|scriptname")]

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

Parameters

portname:	*the name of the port to open*
switches:	*(optional) specifies the setting of the port like the speed of the port 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.*

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 also must provide a scriptname.

Example usage

```
Set Field[gErrorCode,       External("Serial-Open",  "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,
            External("Serial-Open",  gPortName1 & "|"  &
                        gSpeed & " " & gStopBits & " " & gDataBits & " " & gParity &  " "& gFlowControl & "|" &
                        Status(CurrentFileName) &"|"&
                        "Process Data Received for 1st Port"
                        )
]
```

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 SetField[result, External("Serial-Receive" , "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

portname:	*the name of the port to receive data from*

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.

Example usage

Set Field[gResult, External("Serial-Receive" , "Modem port")]

This will receive data from the Modem port. 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

Serial-Receive

In ScriptMaker add the following script steps:

```
Set Field [gTempResultReceived, External("Serial-Receive", 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-RestoreSituation

Syntax SetField[result, External("Serial-RestoreSituation", ""]

Bring the database file that was in front, before the Dispatch Script was called, back to the front.

Parameters

no parameters leave empty for future use.

Returned result

The returned result is an error code:

0	no error

At the moment no other results are returned.

Example usage

Set Field [gErrorCode, External("Serial-RestoreSituation", "")]

Serial-Send

Syntax SetField[result, External("Serial-Receive" , "portname|data")]

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

Parameters

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 send
$$-28	notOpenErr	The port is not open
$$-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.

Example usage

Set Field[gResult, External("Serial-Send" , "Modem port| So long")]

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

Set Field[gResult, External("Serial-Send" , 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 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
gTextToSend	Global, text, can also be a normal text field
gErrorCode	Global, text

In ScriptMaker add the following script steps:

Serial-Send

```
Set Field [gErrorCode, External("Serial-Send",  gPortName & "|" & gTextToSend) ]
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 SetField[result, External("Serial-SetDispatchScript", "filename | scriptID | waitforstring"]

Sets the Dispatch Script to trigger when data is received. If you give an empty parameter "", the Dispatch Script is removed.

Parameters

filename:	*the name of the file with the Dispatch Script,*
scriptID:	*this indicates the script to be triggered.*
waitforstring:	*(optional) wait for this string of characters before triggering a script.*

scriptID can be either scriptkey=x or scriptname=....

scriptkey=x : the key number in the menu of the Dispatch Script. x must be in the range from 0-9.
scriptname=name: the name of the script to trigger.

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.

Example usage

Set Field[gErrorCode, External("Serial-SetDispatchScript",
    Status(CurrentFileName) & "| scriptname=Read Script| waitforstring=OK")]

This will set the Dispatch Script to the script “Read Script” of the current file. The script will not be triggered before the string “OK” is found.

Set Field[gErrorCode, External("Serial-SetDispatchScript", Status(CurrentFileName) & "| scriptkey=1")]

This will set the Dispatch Script to the script with shortcut control-1 (or command-1) of the current file.

Example 2

Set Field[gErrorCode, External("Serial-SetDispatchScript", "")]

This will reset the Dispatch Script. No action is taken when data is received.

Serial-ToASCII

Syntax SetField[result, External("Serial-ToASCII", "ASCIInumber1 | ASCIInumber2 | ASCIInumber3 |...."]

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

Parameters

ASCIInumber(s) one or more numbers in the range from 0-255

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.

Example usage

Set Field [text, External("Serial-ToASCII", "65|65|80|13")]

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

Example 2

Set Field [text, External("Serial-ToASCII", "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-Version

Syntax Set Field [result, External("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)*

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:

VersionString:
If you asked for the version string it will return for example "Serial Plug-in 2.1"

VersionNumber:
If you asked for the version number it returns the version number of the plug-in x1000. For example version 2.2 will return number 2200.

ShowFlashDialogResult:
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 an empty field to any external function that is not loaded.

Example usage

External("Serial-Version", "") will for example return "Serial Plug-in 2.2"

Example 2

External("Serial-Version", "-GetVersionNumber") will return 1100 for version 1.1.
External("Serial-Version", "-GetVersionNumber") will return 1101 for version 1.1b1
External("Serial-Version", "-GetVersionNumber") will return 2130 for version 2.1.3

So for example to use a feature introduced with version 1.3 test if the result is equal or greater than 1300.

Table of Contents

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
°	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
	185	0xB9
	186	0xBA
ª	187	0xBB
º	188	0xBC
	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
·	225	0xE1
‚	226	0xE2
„	227	0xE3
‰	228	0xE4
Â	229	0xE5
	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