TROI SERIAL PLUG-IN™ 2.0 USER GUIDE

October 1999

Troi Automatisering
Vuurlaan 18
2408 NB Alphen a/d Rijn
The Netherlands
Tel: +31-172-426606
Fax: +31-172-470539

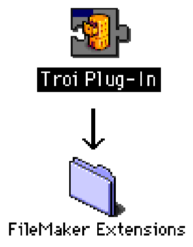
You can also visit the Troi web site at: <http://www.troi.com/> for additional information.

Table of Contents

Installing plug-ins

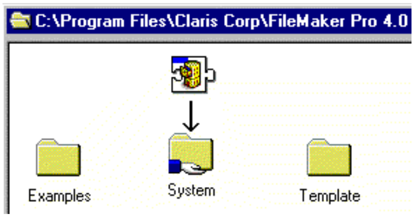
For Macintosh:

- Quit FileMaker Pro.
- Put the file "Troi Serial Plug-in" from the folder "MacOS" 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" 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 "SeriExpl.fp3" to see how to use the plug-in's functions. There is also a Function overview in this file.

IMPORTANT There is a problem in FileMaker Pro 4.0v1. Please make sure that all plug-ins that are in the folder "FileMaker Extensions" are enabled in the preferences. (Under Edit/ Preferences/ Application/ Plug-ins). Make sure all plug-ins have a cross before their name. Remove plug-ins you don't use from the "FileMaker Extensions" folder.
NB: This bug is fixed in version 4.0v2 and later. So please upgrade to the latest versions.

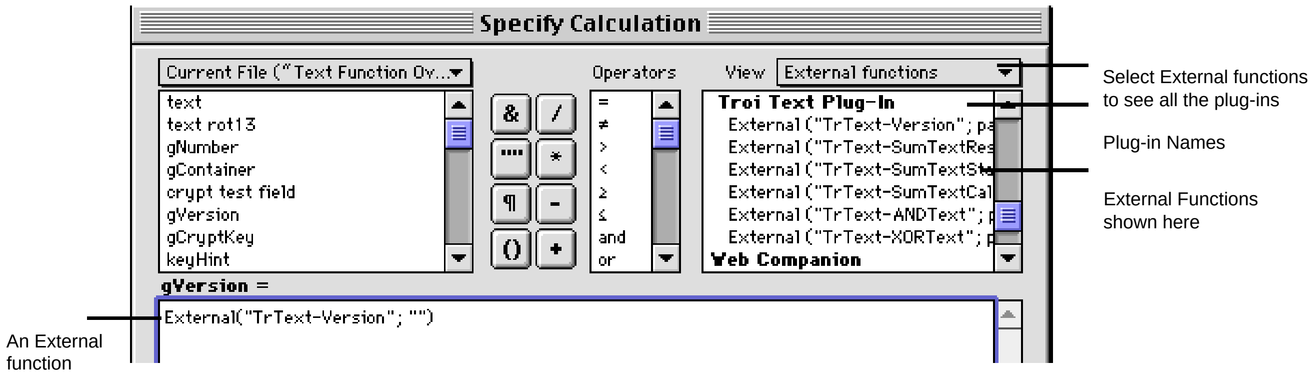
If You Have Problems

This user manual 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!

Summary of functions

Plug-ins add new functions to the standard functions that are available in FileMaker Pro. You can see those extra functions for all plug-ins at the top right of the Specify Calculation Box:


Specify Calculation
Current File ("Text Function Ov...
text
text rot13
gNumber
gContainer
crypt test field
gVersion
gCryptKey
keyHint
Operators
&
/
""
*
¶
-
()
+
=
≠
>
<
≥
≤
and
or
View
External functions
Troi Text Plug-In
External ("TrText-Version"; pa
External ("TrText-SumTextRes
External ("TrText-SumTextSt
External ("TrText-SumTextCal
External ("TrText-ANDText"; p
External ("TrText-XORText"; p
Web Companion
gVersion =
Select External functions to see all the plug-ins
Plug-in Names
External Functions shown here
An External function

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

IMPORTANT In the United States, commas act as list separators in functions. In other countries semi-colons might be used as list separators. The separator being used depends on the operating system your computer uses, as well as the one used when the file was created. All examples show the functions with commas.

The Troi Serial Plug-in adds the following functions:

<u>function name</u>	<u>short description</u>
Serial-Version	check for correct version of the plug-in
Serial-GetPortsNames	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

Using external functions

External functions for this plug-in can be used in a script step using a calculation. The external functions should not be used in a define field calculation.

IMPORTANT The Balance functions have to be used in a specific way, to create the desired effect. See the section on Balance functions for the specifics on this.

Serial-Version

Example usage: External(Serial-Version; "") will return "Troi Serial Plug-in 1.0b1".

IMPORTANT You should always check if the plug-in is loaded, by using this function. If the plug-in is not loaded use of external functions may result in unexpected result or data loss, as FileMaker will return an empty field to any external function that is not loaded.

Serial-GetPortNames

Syntax External("Serial-GetPortNames" , "")

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

Parameters

no parameters	leave empty for future use.

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 desktop Mac a typical result will be:

```
Printer Port¶
Modem Port¶
```

On a portable Mac a typical result will be:

```
Printer-Modem Port¶
Internal Modem¶
```

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 might be returned.

NOTE On Windows currently there is no apparent way to test for the available portnames, so at the moment this function always returns the same result.

Serial-Open

Syntax Set Field[gErrorCode, External("Serial-Open" , "*portname* | *switches* ")]

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.

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	A serial port with this name is not available on this computer
$$-4211	AllPortsNullErr	No serial ports are available on this computer

Other errors might be returned.

Example usage

```
Set Field[ gErrorCode, External("Serial-Open" , "COM2|baud=19200") ]
```

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

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

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=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, 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.

Serial-Close

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

Closes a serial port with the specified name . If the portname parameter is "" ALL ports are closed.

Parameters

portname: the name of the port to close

Result

The returned result is an error code:

0	no error	the port was closed
$$-4210	portDoesnotExistErr	A serial port with this name is not available on this computer
$$-4211	AllPortsNullErr	No serial ports are available on this computer
$$-108	memFullErr	Ran out of memory

Other errors might be returned.

Example Usage

This will close the COM3 port:

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

This will close all open ports:

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

Serial-Receive

Syntax Set Field[gResult, 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

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. See below.

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	A serial 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 might be returned.

Example Usage

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

This will receive data from the Modem port.

Example: Receiving and Testing for Errors

Below you find a "`Receive Data`"script for receiving data into a global text field gTempResultReceived , The script tests for errors. **`gPortName`** is a global text field where the name of the previously opened port was stored.

```
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
End If
```

Serial-Send

Syntax Set Field[gResult, External("Serial-Send" , "*portname* | *data*")]

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

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. See below.

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	portDoesnotExistErr	A serial port with this name is not available on this computer
$$-4211	AllPortsNullErr	No serial ports are available on this computer
$$-207	notEnoughBufferSpace	The output buffer is full

Other errors might be returned.

Example Usage

```
Set Field[ gResult, External("Serial-Send" ,
                   "Modem port| So long and thanks for all the fish") ]
```

This will send the string `"So long and thanks for all the fish"` to the Modem port.

Example: Sending and Testing for Errors

Below you find a "`Send Data`" script for sending data from a global text field **`gTempResultReceived`**, The script tests for errors. **`gPortName`** is a global text field where the name of the previously opened port was stored.

```
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
```

Receiving data via Dispatch Scripting™

FileMaker 5.0 adds 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 uses 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 If you are still using FileMaker 4 on the Windows platform you need to fall back to the original Dispatch Scripting via a key (see below).

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.fp3** for a working example.

Dispatch Scripting using Script Name NEW 2.0

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 cross platform way to execute a script when data has been received, that also works with FileMaker 4.0 on Windows. 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 **NEW 2.0**

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.

Serial-SetDispatchScript

NEW FEATURES 2.0

Syntax Set Field[gResult, External("Serial-SetDispatchScript", "*filename* | scriptID | waitformatch")

Set Field[gResult, External("Serial-SetDispatchScript", "*filename* | scriptkey=*x* ") or
Set Field[gResult, External("Serial-SetDispatchScript", "*filename* | scriptname=*nnnn* ") or
Set Field[gResult, External("Serial-SetDispatchScript", "")

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 which script is to be triggered. See below for details
waitformatch:	(optional) wait for this string of characters before triggering a script. The match-string can be maximum 25 characters long.

The parameter **scriptID** can be one of these forms

scriptname=*nnnn :*	the name of the script to trigger. Not available for FileMaker 4. under Windows.
scriptkey=*x :*	the key number in the menu of the Dispatch Script. *x* must be in the range from 0-9

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 might be returned.

Example Usage

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

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

Example Usage

```
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 Usage (resetting the Dispatch Script)

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

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

Serial-DataWasReceived

Syntax Set Field[gResult, 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.

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-RestoreSituation

Syntax Set Field[gResult, 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.

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-ToASCII

Syntax Set Field[gResult, External("Serial-ToASCII", "asciiCode | asciiCode | asciiCode |...")]

Converts (one or more) numbers to their equivalent ASCII characters. See also Appendix A for a ASCII Table.

Parameters

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

Result

The returned result is the string of text of the ASCII codes.

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.

NOTE You can also use hexadecimal notation for the numbers. Use 0x00 to 0xFF to indicate hexadecimal notation.

Example Usage

```
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.

NOTE The graphic rendition of characters greater than 127 is undefined in the **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange (ASCII Standard) and varies from font to font and from computer to computer and may look different when printed.

Serial-Control NEW 2.0

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

Controls the serial port with the specified name . You can suspend or resume the incoming data with this command. The port needs to be open(See also Serial-Open). This command is very useful for devices that send out continuous data, like an electronic weighing scale. See the example below.

Parameters

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

The *switch* parameter can be either:

suspend	This will suspend reading the incoming stream of data.
resume	This will resume reading the incoming stream of data.

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.

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 receiving by testing if the first two characters are dollars. See below.

Returned error codes can be:

0	noErr	no error
$$-28	notOpenErr	The port is not open
$$-108	memFullErr	Ran out of memory
$$-50	paramErr	There was an error with the parameter
$$-4210	portDoesnotExistErr	A serial port with this name is not available on this computer
$$-4211	AllPortsNullErr	No serial ports are available on this computer

Other errors might be returned.

Example Usage

```
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

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(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`** .

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	Char	Dec	Hex	Char	Dec	Hex
a	97	0x61	°	161	0xA1	·	225	0xE1
b	98	0x62	¢	162	0xA2	‚	226	0xE2
c	99	0x63	£	163	0xA3	„	227	0xE3
d	100	0x64	§	164	0xA4	‰	228	0xE4
e	101	0x65	•	165	0xA5	Â	229	0xE5
f	102	0x66	¶	166	0xA6	Ê	230	0xE6
g	103	0x67	ß	167	0xA7	Á	231	0xE7
h	104	0x68	®	168	0xA8	Ë	232	0xE8
i	105	0x69	©	169	0xA9	È	233	0xE9
j	106	0x6A	™	170	0xAA	Í	234	0xEA
k	107	0x6B	´	171	0xAB	Î	235	0xEB
l	108	0x6C	¨	172	0xAC	Ï	236	0xEC
m	109	0x6D	≠	173	0xAD	Ì	237	0xED
n	110	0x6E	Æ	174	0xAE	Ó	238	0xEE
o	111	0x6F	Ø	175	0xAF	Ô	239	0xEF
p	112	0x70	∞	176	0xB0		240	0xF0
q	113	0x71	±	177	0xB1	Ò	241	0xF1
r	114	0x72	≤	178	0xB2	Ú	242	0xF2
s	115	0x73	≥	179	0xB3	Û	243	0xF3
t	116	0x74	¥	180	0xB4	Ù	244	0xF4
u	117	0x75	µ	181	0xB5	ı	245	0xF5
v	118	0x76	∂	182	0xB6	ˆ	246	0xF6
w	119	0x77	∑	183	0xB7	˜	247	0xF7
x	120	0x78	∏	184	0xB8	¯	248	0xF8
y	121	0x79	π	185	0xB9	˘	249	0xF9
z	122	0x7A	∫	186	0xBA	˙	250	0xFA
{	123	0x7B	ª	187	0xBB	˚	251	0xFB
|	124	0x7C	º	188	0xBC	¸	252	0xFC
}	125	0x7D	Ω	189	0xBD	˝	253	0xFD
~	126	0x7E	æ	190	0xBE	˛	254	0xFE
Del	127	0x7F	ø	191	0xBF	ˇ	255	0xFF
Ä	128	0x80	¿	192	0xC0			
Å	129	0x81	¡	193	0xC1			
Ç	130	0x82	¬	194	0xC2			
É	131	0x83	√	195	0xC3			
Ñ	132	0x84	ƒ	196	0xC4			
Ö	133	0x85	≈	197	0xC5			
Ü	134	0x86	∆	198	0xC6			
á	135	0x87	«	199	0xC7			
à	136	0x88	»	200	0xC8			
â	137	0x89	…	201	0xC9			
ä	138	0x8A		202	0xCA			
ã	139	0x8B	À	203	0xCB			
å	140	0x8C	Ã	204	0xCC			
ç	141	0x8D	Õ	205	0xCD			
é	142	0x8E	Œ	206	0xCE			
è	143	0x8F	œ	207	0xCF			
ê	144	0x90	–	208	0xD0			
ë	145	0x91	—	209	0xD1			
í	146	0x92	“	210	0xD2			
ì	147	0x93	”	211	0xD3			
î	148	0x94	‘	212	0xD4			
ï	149	0x95	’	213	0xD5			
ñ	150	0x96	÷	214	0xD6			
ó	151	0x97	◊	215	0xD7			
ò	152	0x98	ÿ	216	0xD8			
ô	153	0x99	Ÿ	217	0xD9			
ö	154	0x9A	⁄	218	0xDA			
õ	155	0x9B	¤	219	0xDB			
ú	156	0x9C	‹	220	0xDC			
ù	157	0x9D	›	221	0xDD			
û	158	0x9E	ﬁ	222	0xDE			
ü	159	0x9F	ﬂ	223	0xDF			
†	160	0xA0	‡	224	0xE0			