

**FieldCommander<sup>®</sup>**

## FCscript Programmer's Guide

FC-SW

## Table of contents

<b>About this manual</b> .....	<b>4</b>
<b>1. Introduction</b> .....	<b>5</b>
1.1 FCscript and Javascript .....	5
<b>2. Function reference</b> .....	<b>6</b>
2.1 Data processing .....	6
2.1.1 Port .....	6
2.1.2 Buffer .....	11
2.1.3 Filter .....	16
2.1.4 Gate .....	17
2.1.5 Data Trigger .....	18
2.1.6 ASCII Frame Converter .....	19
2.1.7 Export .....	20
2.1.8 Import .....	21
2.1.9 Viewpoint .....	21
2.2 Conditions and expressions .....	22
2.2.1 Conditions .....	22
2.2.2 Expressions .....	24
2.3 Timers .....	24
2.3.1 Interval timer .....	25
2.3.2 Clock timer .....	25
2.4 Event management .....	27
2.4.1 Sources and types .....	27
2.4.2 Event related functions .....	28
2.4.3 Event callbacks .....	29
2.5 Data exchange .....	29
2.5.1 Send data patterns .....	29
2.5.2 Send e-mail .....	30
2.5.3 Send/receive files .....	32
2.5.4 Export as XML .....	34
2.6 Meta tags .....	35
2.7 TCP communication .....	36
2.7.1 Setting up the link .....	36
2.7.2 Establish a connection .....	39
2.7.3 Sending data .....	39
2.8 Database access .....	42
2.8.1 Access from script .....	42
2.8.2 Access the database .....	44
2.8.3 Access by SQL queries .....	48
2.9 Modem functions .....	50
2.9.1 Call setup .....	50
2.9.2 Send text message .....	51
2.9.3 Send multimedia message .....	52
2.10 Audio functions .....	53
2.10.1 Play sound .....	53
2.10.2 Detect DTMF tones .....	54
2.11 Imaging .....	55
2.12 Logging .....	56
2.13 Calling PHP .....	57

---

2.13.1	Passing arguments . . . . .	57
2.13.2	Example . . . . .	57
2.14	System functions . . . . .	59
<b>Function index</b>	. . . . .	<b>60</b>
<b>Notice to the user</b>	. . . . .	<b>61</b>
Software License Agreement	. . . . .	61
Third-party software	. . . . .	62
Trademarks	. . . . .	63
Copyrights	. . . . .	63

## About this manual

This document is intended for application developers and explains the steps to create scripts for FieldCommander. The guide provides all the detail you need to set up your applications. It includes extensive documentation on the syntax and use of FieldCommander's script API. This manual serves both as tutorial and reference guide.

Although it is targeted towards people who are not necessarily experienced in programming, it only touches the generic concept of developing software. When you haven't written macros or scripts before, you might want to pick up a book on programming too.

### Organisation of the documentation

- Installation**      *Installation Guide Hardware Platform (FCHWP<N>IG)*  
Bundles information related to the hardware, such as the installation of the device, IP networking, serial interface details and technical specifications.
- Employment**      *User's Guide (FCSWUG)*  
Explains how FieldCommander is used in your application development, and documents all its features, including the databases, communication and web based configuration.
- Programming**      *FCscript Programmer's Guide (FCSCRIPTPG)*  
Function reference of FieldCommander's scripting language, used to control the data flow and event management in your applications.
- FCphp Programmer's Guide (FCPHPPG)*  
Function descriptions of the FieldCommander specific PHP interface, used to build dynamic user interfaces or reports on top of the your application.
- Javascript Reference Manual (FCJSREF)*  
Extensive documentation of the Javascript (ECMAScript) core language used by FieldCommander for application programming.
- Software options**      The optionally available "plug in" software modules have their own documentation to explain the features, installation, configuration and programming interfaces.

# 1. Introduction

## 1.1 FCscript and Javascript

The scripting language is a core component of FieldCommander as it is used to implement your applications. It's based on the well-known Javascript language though it has been extended with a large set of functions unique to FieldCommander.

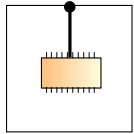
We will refer to FieldCommander's scripting language as **FCscript** to set it apart from the common Javascript. Where this documentation mentions **Javascript**, it refers to features or functions available in the language as officially standardized by the European Computer Manufacturers Association (ECMA) as ECMA-262 or ECMAScript.

FCscript is not limited to Javascript alone. It adds functionality to read and write files on its disk and a vast number of functions to control the data processing, communications and event handling of FieldCommander.

## 2. Function reference

### 2.1 Data processing

#### 2.1.1 Port



**Name:** LocalPort  
**Purpose:** interface driver for physical port, the source of data units  
**Input:** raw serial data  
**Output:** stream of data units

The LocalPort component provides the interface between the physical serial ports and the data streams used in FieldCommander. Raw serial frames enter the port where they are converted into a usable stream of data units, which can be processed by the next component in the network.

#### **newLocalPort()**

**SYNTAX:** `portID = newLocalPort(port number, data type [, interface]);`

**WHERE:** port number - the hardware port number as indicated on the FieldCommander unit  
 data type - see table  
 interface - (optional) see table

port usage	data type	interface
plain serial RS-232	DT_RS	IF_RS (default)
plain serial RS-485	DT_RS	IF_RS (default)
EIB/KNX <sup>1)</sup>	DT_AF	IF_EIBBCU1
USB/serial	DT_RS	IF_USB_RS

<sup>1)</sup> EIB/KNX support available as an option

**RETURN:** portID - port identifier on success, -1 on failure

**DESCRIPTION:** Creates and initiates the given hardware port with default values and assigns it a portID. Use setLocalPort() to set your custom values. Only one instance of each hardware port can be created.

**EXAMPLES:**

```
// create a plain RS-232 port
rsPort = newLocalPort(1, DT_RS);

// create a port for a EIB/KNX interface in 2nd port
eibPort = newLocalPort(2, DT_AF, IF_EIBBCU1);

// create a port for a USB based DMX interface
dmxPort = newLocalPort(1, DT_RS, IF_USB_RS);
```

**newAudioPort()**

SYNTAX: `portID = newAudioPort([port number]);`

WHERE: `port number` - (optional) the audio number, can be omitted when just one audio interface is available

RETURN: `portID` - port identifier on success, -1 on failure

DESCRIPTION: Creates and initiates the audio interface port with default values and assigns it a portID.

Use `setLocalPort()` to load your custom settings. This function is identical to setting up a `newLocalPort` with interface `IF_AUDIO` and datatype `DT_RS`.

EXAMPLES:

```
// create an audio port
audioPort = newAudioPort();

// create a port for audio interface 2
audioPort = newAudioPort(2);
```

**newModemPort()**

SYNTAX: `portID = newModemPort(port number [, mode]);`

WHERE: `port number` - the hardware port number as indicated on the FieldCommander unit  
`mode` - (optional) see table

Modes for modem interface:

mode	description
MODEM_STANDARD (default)	plain (analog) modem with dial functions according V.25ter standard
MODEM_NULL	direct serial cable connection (NULL modem wiring)
MODEM_GSM	GSM mobile modem with SMS capabilities according GSM 07.05 and 07.07 standards
MODEM_GSM_GPRS	GSM mobile modem with SMS + GPRS capabilities according GSM 07.05 and 07.07 standards
MODEM_SIEMENS_MC35	special mode for MC35 modem of Siemens

RETURN: `portID` - port identifier on success, -1 on failure

DESCRIPTION: Creates and initiates the given modem port with default values and assigns it a portID.

Use `setLocalPort()` to load your custom settings. This function is identical to setting up a `newLocalPort` with interface `IF_MODEM` and datatype `DT_SMS`.

EXAMPLES:

```
// create a port for GSM/GPRS capable modem
modemPort = newModemPort(1, MODEM_GSM_GPRS);

// create a port for a Siemens MC35 modem
mc35Port = newModemPort(1, MODEM_SIEMENS_MC35);
```

**setLocalPort()**

SYNTAX: `setLocalPort(portID, parameter, value);`

WHERE: portID - port identifier returned by newLocalPort()  
 parameter - see tables below for possible settings  
 value - depends on parameter, see tables below for possible values

**For serial interface (IF\_RS, default):**

parameter	value	default
RS_BAUDRATE	50 ~ 115.200 <sup>1)</sup>	9600
RS_STOPBITS	1 or 2	1
RS_DATABITS	5, 6, 7 or 8	8
RS_PARITY	RS_PARITY_ODD, RS_PARITY_EVEN, RS_PARITY_NONE, RS_PARITY_MARK or RS_PARITY_SPACE	RS_PARITY_NONE
RS_RECEIVEFIFO	0 ~ 14 <sup>1)</sup>	8
RS_FLOWCONTROL <sup>4)</sup>	RS_FLOW_NONE, RS_FLOW_XONXOFF, RS_FLOW_RTSCS, <sup>3)</sup> RS_FLOW_DTRDSR <sup>3)</sup>	RS_FLOW_NONE

<sup>1)</sup> HWP10 - except port 1 - supports rates up to 230.400 bps (RS232) / 1.500.00 (RS485)

<sup>2)</sup> HWP10 - except port 1 - supports FIFO up to 127 bytes deep (default: 32)

<sup>3)</sup> Port 1 of HWP10 does not support RTS/CTS and DTR/DSR flow control

<sup>4)</sup> RS485 ports have automatic flow control so this parameter is not applicable

**For modem interface (created with newModemPort):**

parameter	description
MODEM_EXTRINIT	Optional extra initialization of the modem through AT commands. Max 256 characters. The line termination by CR and/or LF can be omitted.

Additional for modem interface with SMS capabilities:

parameter	description
MODEM_PIN	PIN code of SIM card in modem, use empty string or omit when no PIN code protection is enabled
MODEM_SMSCENTER	Message center's phone number of the SIM's provider, in international notation (eg. +316540881000)

Additional for modem interface with GPRS capabilities:

parameter	description
MODEM_APN	Access point name/address at provider
MODEM_USER	APN user name, set empty string or omit when no authentication is used
MODEM_PASSWORD	APN password, set empty string or omit when no authentication is used

Additional for modem interface with MMS capabilities:

parameter	description
MODEM_WAPADDRESS	Gateway (WAP) IP address and -optionally- port number
MODEM_MMSCURL	MMSC homepage URL

These details should be supplied by the provider of your SIM card.

**For audio interface (created with newAudioPort):**

parameter	value	default	description
AUDIO_VOL_OUT	0 ~ 100	50	audio output volume
AUDIO_VOL_IN	0 ~ 100	0	line input level
AUDIO_VOL_MIC	0 ~ 100	0	microphone input level
AUDIO_INPUT	AUDIO_LINE, AUDIO_MIC	AUDIO_LINE	input channel selection

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Configure a setting for the given hardware interface. Available parameters and values are device dependent, see tables above.

NOTE: FieldCommander supports all possible values for the baudrate within the specified range. The real baudrate may vary from the baudrate set with setLocalPort(), the maximum deviation is two percent of the defined value. The real baudrate value can be obtained with getLocalPort().

EXAMPLES:

```
// setup port 1 to 9600 baud, 8 databits, 1 stopbit, no parity,
// fifolevel 1, no flowcontrol
rsPort = newLocalPort(1, DT_RS);
setLocalPort(rsPort, RS_BAUDRATE, 9600);
setLocalPort(rsPort, RS_DATABITS, 8);
setLocalPort(rsPort, RS_STOPBITS, 1);
setLocalPort(rsPort, RS_PARITY, RS_PARITY_NONE);
setLocalPort(rsPort, RS_TRANSMITFIFO, 1);
setLocalPort(rsPort, RS_RECEIVEFIFO, 1);
setLocalPort(rsPort, RS_FLOWCONTROL, RS_FLOW_NONE);
```

```
// setup modem at port 2 for both SMS and MMS messaging through Vodafone
smsPort = newModemPort(2);
setLocalPort(smsPort, MODEM_PIN, "0000");
setLocalPort(smsPort, MODEM_SMSCENTER, "+316540881000");
setLocalPort(smsPort, MODEM_APN, "live.vodafone.com");
setLocalPort(smsPort, MODEM_USER, "vodafone");
setLocalPort(smsPort, MODEM_PASSWORD, "vodafone");
setLocalPort(smsPort, MODEM_MMSCURL, "http://mmsc.mms.vodafone.nl");
setLocalPort(smsPort, MODEM_WAPADDRESS, "192.168.251.150:9102");

// setup audio interface with microphone as input
audioPort = newAudioPort();
setLocalPort(audioPort, AUDIO_VOL_OUT, 100);
setLocalPort(audioPort, AUDIO_VOL_IN, 0);
setLocalPort(audioPort, AUDIO_VOL_MIC, 40);
setLocalPort(audioPort, AUDIO_INPUT, AUDIO_MIC);
```

---

### **getLocalPort()**

**SYNTAX:** value = getLocalPort(portID, parameter);

**WHERE:** portID - port identifier returned by newLocalPort()  
parameter - device dependent

**RETURN:** value - depends on parameter, -1 on failure

**DESCRIPTION:** Request the current setting of the given port. Available parameters and returned values are device dependent, see the tables at setLocalPort(). Not all settings can be read back.

**EXAMPLE:**

```
// Get port 1 settings and write the values to the log
rsPort = newLocalPort(1, DT_RS);
baudrate = getLocalPort(rsPort, RS_BAUDRATE);
databits = getLocalPort(rsPort, RS_DATABITS);
stopbits = getLocalPort(rsPort, RS_STOPBITS);
parity = getLocalPort(rsPort, RS_PARITY);
writeLog("Port baudrate: " + baudrate);
writeLog("Port format : " + databits + "," + stopbits + "," + parity );
```

---

### **openPorts()**

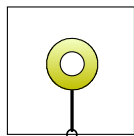
**SYNTAX:** openPorts();

**RETURN:** number - 0 on success, 1 on failure

**DESCRIPTION:** Starts the capturing of data from the created ports. This function is typically called at the end of the main() function, after all initialization and configuration is completed. In most cases it is followed by the sleep() command, after which all script processing goes by handling of events.

---

### 2.1.2 Buffer



**Name:** Buffer  
**Purpose:** store and serve data units  
**Input:** stream of data units  
**Output:** none

The Buffer component is used to store and serve data units. Once data units are stored, you can browse and search the buffer for data in elements, and fetch it for further processing. Examples are searching for certain data strings, values, flags or time stamps. Typically, this is done in the `handleEvent()` routine of a script.

You can configure the Buffer's capacity and wrapping mode. In *linear mode*, data is added to the buffer until it is full. Once the buffer has reached its maximum capacity, it triggers an event *evBufferFull*, so you can take responsive actions in the script. In *ring mode*, the buffer is never full. It just overwrites the oldest data units, replacing it with the most recent data units in the stream. Refer to Event Management for more information on the *evBufferFull* event.

#### **addBuffer()**

---

SYNTAX: `bufferID = addBuffer(sourceID, "name", size, mode);`

WHERE: `sourceID` - ID of the component providing the stream data units  
`name` - name that can be used to refer to the buffer  
`size` - buffer size in number of data units: 10-100000  
`mode` - select either ring or linear mode: RING or LINEAR

RETURN: `bufferID` - buffer identifier on success, -1 on failure

DESCRIPTION: Add a Buffer component to the data stream. RING results in a wrapping buffer. The buffer wraps automatically and overwrites the oldest data. LINEAR results in a linear buffer.

When the buffer is in LINEAR mode an event *evBufferFull* is generated when the end of the buffer is reached.

A system meta tag is created for each buffer containing a value that represents the number of data units that are currently in the buffer. The system meta tag is named `_NDU<name>` where `<name>` stands for the defined name of the buffer.

Note: The maximum Buffer size is limited to 100000 data units. There is however a practical limit to the buffer size, since Buffers are created and maintained in FieldCommander's main memory area. Depending on the data unit type and size and the total number of buffers in use, you could theoretically run out of main memory space.

---

#### **clearBuffer()**

SYNTAX: `clearBuffer(bufferID);`

WHERE: `bufferID` - the buffer identifier

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Delete the buffer content.

---

**gotoFirst()**

---

SYNTAX: `gotoFirst(bufferID);`

WHERE: `bufferID` - the `bufferID` identifier

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Set the pointer to the first data unit in the given data buffer. Use `getBufferDataElement()` to retrieve information from a data unit.

---

**gotoNext()**

---

SYNTAX: `gotoNext(bufferID [, count]);`

WHERE: `bufferID` - the buffer identifier  
`count` - (optional) the number of data units to move the pointer (1 or higher), when omitted, a count of 1 is assumed

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Move the pointer `count` positions forward in the given data buffer. If the pointer is undefined and there is data in the buffer, the first call of `gotoNext()` will set the pointer to the first data unit of the buffer. When there is no data in the buffer, or when the end of the buffer is reached the function will return failure. Use `getBufferDataElement()` to retrieve information from a data unit.

EXAMPLE: 

```
// read all data from the start of the buffer to the end and write
// the values to the log
while( !gotoNext(bufferID, 1) ) {
    data = getBufferDataElement(bufferID, DT_AF, "data");
    writeLog("data:" + data);
}
```

---

**gotoLast()**

---

SYNTAX: `gotoLast(bufferID);`

WHERE: `bufferID` - the buffer identifier

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Set the pointer to the last data unit of the given data buffer. Use `getBufferDataElement()` to retrieve information from a data unit.

---

**gotoPrevious()**

SYNTAX: `gotoPrevious(bufferID [, count]);`

WHERE: `bufferID` - the buffer identifier  
`count` - (optional) the number of data units to move the pointer (1 or higher), when omitted, a count of 1 is assumed

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Move the position *count* positions backward in the given data buffer. If the pointer is undefined and there is data in the buffer, the first call of `gotoPrevious()` will set the pointer to the last data unit of the buffer. When there is no data in the buffer, or when the start of the buffer is reached the function will return failure. Use `getBufferDataElement()` to retrieve information from a data unit.

EXAMPLE: 

```
// read all data from the end of the buffer to the start and write
// the values to the log
while( !gotoPrevious(bufferID, 1) ) {
    data = getBufferDataElement(bufferID, DT_AF, "data");
    writeLog("data:" + data);
}
```

**findValue()**

SYNTAX: `findValue(bufferID, data unit type, "element", value);`

WHERE: `bufferID` - the buffer identifier  
`data unit type` - data unit type of the data in the buffer  
`element` - string, element name of the data unit type, element must contain a value  
`value` - value to search for

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Search the buffer for a matching value from the current position towards the end of the buffer. If a match is found, the pointer is set to the corresponding data unit in the buffer. If the pointer is undefined or the end of buffer is reached without a match, the function will return failure.

Use `getBufferDataElement()` to get the requested data. Use `gotoFirst()`, `gotoNext()`, `gotoLast()` or `gotoPrevious()` to initialize.

**findString()**

SYNTAX: `findString(bufferID, data unit type, "element", "string", options);`

WHERE: `bufferID` - the buffer identifier  
`data unit type` - data unit type of the data in the buffer  
`element` - string, element name of the data unit type, element must contain a string  
`string` - string to search for  
`options` - optional options for the search, may be omitted

<i>options</i>	<i>value</i>
CASE_SENSITIVE	default, search case sensitive
IGNORE_CASE	ignore case
SUB_STRING	default, a match is found when only a part of a the string of an element matches the provided string
WHOLE_STRING	a match is found if only the whole string of an element matches the provided string

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Search the buffer for a matching string starting from the current position towards the end of the buffer. If a match is found, the pointer is set to the corresponding data unit in the buffer. If the pointer is undefined or the end of buffer is reached without a match, the function will return failure.

With the options variable it is possible to define the way the search is executed. If no options are supplied the default options apply. To specify several options use the '|' sign to separate them.

Use `getBufferDataElement()` to get the requested data. Use `gotoFirst()`, `gotoNext()`, `gotoLast()` or `gotoPrevious()` to initialize the pointer for the first time.

### **findFlags()**

SYNTAX: `findFlags(bufferID, data type, "element", flags_high, flags_low);`

WHERE: `bufferID` - the buffer identifier  
`data type` - data unit type of the data in the buffer  
`element` - string, element name of the data unit type, element must contain flags  
`flags_high` - bit mask for flags that should be high to match this search  
`flags_low` - bit mask for flags that should be low to match this search

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Searches the given buffer for a matching flag status pattern, starting from the current position towards the end of the buffer. If a match is found, the pointer is set to the corresponding data unit in the buffer. If the pointer is undefined or the end of buffer is reached without a match, the function will return failure.

The data in the selected element is interpreted as a binary value. Flags only exist for the DT\_RS data type.

Flags for this data type can be:

RS_DTR	: DTR control line
RS_RTS	: RTS control line
RS_CTS	: CTS control line
RS_OE	: Overrun Error
RS_PE	: Parity Error
RS_FE	: Frame Error
RS_BRK	: Break Error
RS_DCD	: DCD control line
RS_RI	: RI control line
RS_DSR	: DSR control line
RS_ERROR	: Any error occurred (OE, PE, FE or BRK).

To specify several controls for the high or low mask use the '|' sign to separate them. Use `getBufferDataElement()` to get the requested data. Use `gotoFirst()`, `gotoNext()`, `gotoLast()` or `gotoPrevious()` to initialize the pointer for the first time.

EXAMPLE: 

```
// find position where RTS is high, DTR is high and Error is low
findFlag( bufferID, DT_RS, "control", RS_RTS|RS_DTR, RS_ERR );
```

**findTime()**

SYNTAX: `findTime(bufferID, data type, "element", timestamp, window);`

WHERE: `bufferID` - the buffer identifier  
`data type` - data unit type of the data in the buffer  
`element` - string, element name of the data unit type, element must contain time  
`timestamp` - timevalue to search for  
`window` - timevalue with the relaxing time span around the time to search

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Searches the given buffer for the best match of the timestamp within the period given by window. The search is started from the current position towards the end of the buffer. If a match is found, the pointer is set to the corresponding data unit in the buffer. If the pointer is undefined or the end of buffer is reached without a match, the function will return failure. When *window* is 0, findTime() will only be successful in case an exact match is found.

Use `getBufferDataElement()` to get the requested data. Use `gotoFirst()`, `gotoNext()`, `gotoLast()` or `gotoPrevious()` to initialize the pointer for the first time.

**getBufferDataElement()**

SYNTAX: `getBufferDataElement(bufferID, data type, "element");`

WHERE: `bufferID` - the buffer identifier  
`data type` - data unit type of the data in the buffer  
`element` - string, element name of the data unit type

RETURN: value from the given data element on the current position in the given buffer

DESCRIPTION: Get the data element from the data buffer at the current position. The position should be set by calling goto... or find... functions.

EXAMPLE: 

```
// get a value from a buffer containing the DT_RS data type
// data will contain a value
data = getBufferDataElement( bufferID1, DT_RS, "data" );

// get flags from a buffer containing the DT_RS data type
// data will contain a value
data = getBufferDataElement( bufferID1, DT_RS, "control" );

// get a string from a buffer containing the DT_AF data type
// data will contain a string
data = getBufferDataElement( bufferID2, DT_AF, "data" );

// get a time stamp from a buffer containing the DT_AF data type
// data.sec will contain the major part of the timeval in seconds
// data.usec will contain the minor part of the timeval in microseconds
data = getBufferDataElement( bufferID2, DT_AF, "timestamp" );
```

**getBufferDataUnit()**

SYNTAX: `getBufferDataUnit(bufferID, data type);`

WHERE: `bufferID` - the buffer identifier  
`data type` - data unit type of the data in the buffer

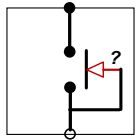
RETURN: Object from the given data unit on the current position in the given buffer

**DESCRIPTION:** Get the complete data unit from the data buffer at the current position. The position should be set by calling goto... or find... functions.

**EXAMPLE:**

```
// get a value from a buffer containing the DT_RS data type
dataunit = getBufferDataUnit( bufferID1, DT_RS);
writeLog("Data byte: "+dataunit.data);
writeLog("Time/date: "+Date.fromSystem(dataunit.timestamp.sec).toString());
```

### 2.1.3 Filter



**Name:** Filter  
**Purpose:** select data unit  
**Input:** stream of data units  
**Output:** selected stream of data units

The Filter component is used to select single data units. Only data units matching the - combination of - condition(s) are passed to the next components in the network. Other data units are dropped.

Conditions can be defined on any combination of values, string matches, times or bit states. Refer to Conditions and expressions for details on conditions.

#### addFilter()

**SYNTAX:** filterID = addFilter(sourceID, "expression", mode);

**WHERE:** sourceID - the source identifier  
 expression - boolean expression on conditions using conditionID's  
 mode - select either REPEAT or ONCE

**RETURN:** filterID - the identifier on success, -1 on failure

**DESCRIPTION:** Select single data units from the stream of data units to pass to a next component in the network. Use an expression to set condition matching rules. In ONCE mode, the filter will only work a single on the time on the first expression match it encounters and needs to be reset in order to be used again (see resetFilter()). When working in REPEAT mode, a Filter will continuously work on every expression match.

**EXAMPLE:**

```
portID = newLocalPort(1, DT_RS);

// pass all data starting except character "a" (96 decimal)
condID1 = setValueCondition( DT_RS, "data", 96, NOT_EQUAL );
addFilter( portID, "condID1", REPEAT );
```

#### resetFilter()

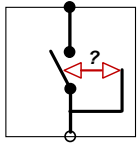
**SYNTAX:** resetFilter(filterID);

**WHERE:** filterID - the filter identifier

**RETURN:** number - 0 on success, 1 on failure

**DESCRIPTION:** Reset the target filter to it's default state. Applies only to a filter that is working in ONCE mode.

### 2.1.4 Gate



**Name:** Gate  
**Purpose:** select a stream of data units  
**Input:** stream of data units  
**Output:** selected stream of data units

The Gate component can be used to select partial streams of data units in a stream to pass on to the next component in the network. It uses start and stop conditions to open and close the gate, respectively. A closed gate drops all data units until a matching expression is detected to open the gate. When the gate opens, the data units are passed to the next component(s) in the network, until the expression to close the gate is matched. The data units causing the gate to open and close are also passed.

Conditions can be defined on any combination of values, string matches, times or bit states. Refer to Conditions and expressions for details on conditions.

#### addGate()

**SYNTAX:** `gateID = addGate(sourceID, "start_expression", "stop_expression", mode);`

**WHERE:** sourceID - the source identifier  
start\_expression - boolean expression on conditions using conditionID's  
stop\_expression - boolean expression on conditions using conditionID's  
mode - select either REPEAT | ONCE

**RETURN:** gateID - the gate identifier on success, -1 on failure

**DESCRIPTION:** Select partial streams of data units to pass to a next component in the network. Use start and stop expressions to open or close the gate. In ONCE mode, the gate will only work a single time on the first expression match it encounters and needs to be reset in order to be used again (see resetGate()). When working in REPEAT mode, a Gate will continuously work on every expression match.

**EXAMPLE:**

```
portID = newLocalPort(1, DT_RS);

// pass all data starting with character "a" (96 decimal) until a
// character "b" (97 decimal) is detected
condID1 = setValueCondition(DT_RS, "data", 96, EQUAL);
condID2 = setValueCondition(DT_RS, "data", 97, EQUAL);
addGate(portID, "condID1", "condID2", REPEAT);
```

#### resetGate()

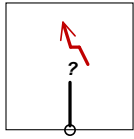
**SYNTAX:** `resetGate(gateID);`

**WHERE:** gateID - the gate identifier

**RETURN:** number - 0 on success, 1 on failure

**DESCRIPTION:** Reset target gate to it's default state. Applies only to a gate that is working in ONCE mode.

### 2.1.5 Data Trigger



**Name:** DataTrigger  
**Purpose:** generate an event on condition match  
**Input:** stream of data units  
**Output:** none

The Data Trigger is used to generate an event when one or more elements in a data unit match the specified condition. The generated event is handled by the Event Manager so you can take responsive action in the script. Typically, this is done in the `handleEvent()` routine.

Conditions can be defined on any combination of values, string matches, times or bit states. Refer to "Conditions and expressions" for details on conditions, and the chapter "Event management" on how to set up the script.

#### **addDataTrigger()**

---

**SYNTAX:** `datatriggerID = addDataTrigger(sourceID, "expression", mode);`

**WHERE:** `sourceID` - the source component identifier  
`expression` - boolean expression on conditions using conditionID's.  
`mode` - select either REPEAT or ONCE

**RETURN:** `datatriggerID` - the datatrigger identifier on success, -1 on failure

**DESCRIPTION:** Trigger an event when the supplied expression of conditions is matched in the data stream. The event will be handled in the `handleEvent()` routine in the script. In ONCE mode, the Data Trigger will only work a single time on the first expression match it encounters and needs to be reset in order to be used again (see `resetDataTrigger()`). When working in REPEAT mode, a data trigger will continuously work on every expression condition match.

**EXAMPLE:**

```
portID = newLocalPort(1, DT_RS);

// set a data trigger on each occurrence of character "a" (96 decimal)
condID1 = setValueCondition(DT_RS, "data", 96, EQUAL);
addDataTrigger(portID, "condID1", REPEAT);
```

---

#### **resetDataTrigger()**

---

**SYNTAX:** `resetDataTrigger(datatriggerID);`

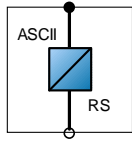
**WHERE:** `datatriggerID` - the data trigger identifier

**RETURN:** `datatriggerID` - the datatrigger identifier on success, -1 on failure

**DESCRIPTION:** Reset the target data trigger to it's original state. Applies only to a data trigger that is working in ONCE mode.

---

### 2.1.6 ASCII Frame Converter

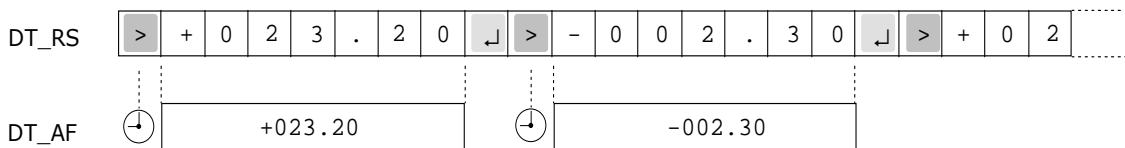


**Name:** converter, type: ASCII\_FRAME  
**Purpose:** convert single bytes into an ASCII string  
**Input:** stream of data units of type DT\_RS  
**Output:** stream of data units of type DT\_AF

In the DT\_RS data type, every byte is represented by a data unit. Most communication protocols use single bytes or strings to indicate the start and end of data frame. The ASCII frame converter is able to convert the stream of data units with single bytes into data units with a complete string frame. You can pass the start & end strings and/or the frame length.

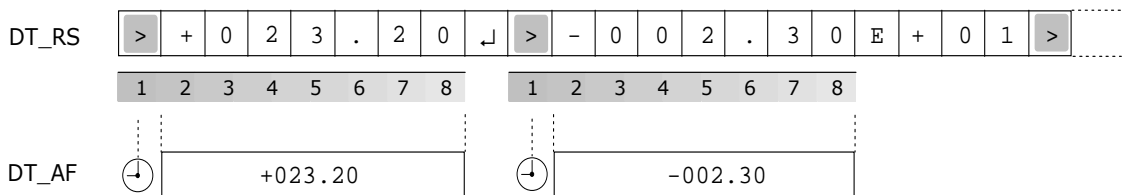
Example 1: start/end flags

start >  
 end ↵  
 length --



Example 2: fixed frame length

start >  
 end --  
 length 8



The resulting ASCII frame data unit (DT\_AF) has a data element that contains a string of characters (length max 256) and a timestamp of the first DT\_RS character of the frame start. The information of the control element is not part of the DT\_AF data unit. You can configure the converter to either include or exclude the start and/or end flags in the ASCII frame.

#### addConverter()

SYNTAX: `converterID = addConverter(sourceID, converter_type, "frame_start", "frame_end", frame_length [, options]);`

WHERE:   
 sourceID - the source identifier  
 converter\_type - defines the converter type: ASCII\_FRAME  
 frame\_start - string defining the start of an ASCII frame, max. 10 characters  
 frame\_end - string defining the end of an ASCII frame, max. 10 characters  
 frame\_length - the number of characters in the frame to be detected (incl. start & end)  
 options - (optional) switches to include or exclude the start/end, see options below

<i>options</i>	<i>description</i>
AF_EXCLUDE_START	(default) the frame_start string is not part of the result
AF_INCLUDE_START	the frame_start string is included in the result
AF_EXCLUDE_END	(default) the frame_end string is not part of the result
AF_INCLUDE_END	the frame_end string is included in the result

ASCII frames are limited to 256 character. Note that the start and end strings, count as characters when evaluating the `frame_length`, even when they are excluded from the resulting frame.

When `frame_start` and `frame_end` expression are the same, the ASCII frame converter will work as a filter: only the data that matches the defined string are processed. The `frame_length` and options are ignored in this case.

The ASCII frame converter can be also used to find end-of-line patterns. When no `frame_start` is given, the `frame_end` determines the end of the frame. A new frame starts immediately after the end of the frame previous one. `Frame_length` is ignored in this case.

RETURN: `converterID` - the converter identifier on success, -1 on failure

DESCRIPTION: Convert a stream of DT\_RS data units into a single DT\_AF data unit with the detected frame as a single string of ASCII characters.

With the options variable it is possible to define the output of the converter. If no options are supplied the default options apply. To specify several options use the '|' sign to separate them.

```
EXAMPLE: portID = newLocalPort(1, DT_RS);

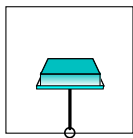
// ascii frame converter that detects frames that start with "ab" and end
// with "cd". The frame length option is ignored because the frame end is
// specified. The frame start and frame end are included in the resulting
// frame
converterID1 = addConverter(portID, ASCII_FRAME, "ab", "cd", 0,
                          AF_INCLUDE_START | AF_INCLUDE_END);

// ascii frame converter that detects frames that start with "ab" and are
// are 8 characters long (including the start_frame, though it is omitted
// in the resulting frame)
converterID2 = addConverter(portID, ASCII_FRAME, "ab", "", 8);

// ascii frame converter that output frames ending with (and including)
// "ab", the frame length is ignored
converterID3 = addConverter(portID, ASCII_FRAME, "", "ab", 0,
                          AF_INCLUDE_END);

// ascii frame converter that output frames "ab", the frame length is
// ignored and options are omitted
converterID4 = addConverter(portID, ASCII_FRAME, "ab", "ab", 0);
```

### 2.1.7 Export



**Name:** Export  
**Purpose:** share information with other FieldCommander devices  
**Input:** stream of data units  
**Output:** stream of data units over TCP/IP

FieldCommander devices can be connected to form a distributed network. The Export component will set up a TCP/IP connection and export all data units in the stream. The FieldCommander you connect to, should support importing streams through a Remote Port component.

**addExport()**


---

SYNTAX: `exportID = addExport(sourceID, "address", port number);`

WHERE: `sourceID` - the source component identifier  
`address` - the IP address of the target FieldCommander unit.  
`port number` - TCP/IP port number that is used to contact the target FieldCommander unit.

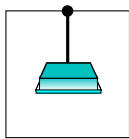
RETURN: `exportID` - export identifier on success, -1 on failure

DESCRIPTION: Exports a captured stream of data units to a (remote) FieldCommander unit over a TCP/IP network connection.

---

**2.1.8 Import**

(in selected models)



**Name:** Remote Port  
**Purpose:** import information from other FieldCommander devices  
**Input:** stream of data units over TCP/IP  
**Output:** stream of data units

FieldCommander devices can be connected to form a distributed network. The Remote Port component accepts a TCP/IP connection from other FieldCommander devices and puts all data units in the stream.

**newRemotePort()**


---

SYNTAX: `portID = newRemotePort(portnumber, datatype);`

WHERE: `portnumber` - TCP/IP port number that is used for remote connections  
`datatype` - the data unit type that the imported stream contains

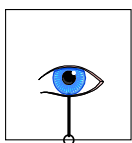
RETURN: `portID` - port identifier on success, -1 on failure

DESCRIPTION: Opens a Remote Port. Other FieldCommander devices can connect over the TCP/IP network to the Remote Port with the Export component.

---

**2.1.9 Viewpoint**

(in selected models)



**Name:** Viewpoint  
**Purpose:** export data from the data stream to the WebGUI  
**Input:** data units  
**Output:** data elements over a TCP/IP connection to the WebGUI

Viewpoints are one of the three possible sources to export data from the FieldCommander to the graphical user interface, referred to as WebGUI. A Viewpoint is created with the function `addViewPoint` and is placed in a stream like all other components.

When the WebGUI applet is started, it registers itself to all its sources, including viewpoints. When a Viewpoint in a stream receives a new data unit, it is sent to all applets that registered themselves to the Viewpoint. For more information on the WebGUI, check chapter 6.

**addViewpoint()**


---

SYNTAX:            converterID = addViewpoint(sourceID, name);

WHERE:            sourceID - the ID of the component that is the data source for the converter  
                     name - name that is used by the WebGUI applet to refer to the Viewpoint

RETURN:           portID - port identifier on success, -1 on failure

DESCRIPTION:    Adds a Viewpoint

---

## 2.2 Conditions and expressions

Conditions and expressions are used to control the Filter, Gate and Trigger components and require additional explanation.

### 2.2.1 Conditions

A condition specifies a data state or occurrence which results in a logical "true" or "false" value.

**setValueCondition()**


---

SYNTAX:            conditionID = setValueCondition(data type, "element", value, options);

WHERE:            data type - type of data unit the condition applies to  
                     element - string, name of data unit element the condition applies to  
                     value - number that is used for the evaluation  
                     options - the type of evaluation that is performed

<i>options</i>	<i>evaluation</i>
EQUAL	is the value of the defined element of the supplied data unit <b>equal to</b> the defined value?
NOT_EQUAL	is the value of the defined element of the supplied data unit <b>not equal to</b> the defined value?
GREATER	is the value of the defined element of the supplied data unit <b>greater than</b> the defined value?
GREATER_OR_EQUAL	is the value of the defined element of the supplied data unit <b>greater than or equal to</b> the defined value?
LESS	is the value of the defined element of the supplied data unit <b>less than</b> the defined value?
LESS_OR_EQUAL	is the value of the defined element of the supplied data unit <b>less than or equal to</b> the defined value?

RETURN:           conditionID - condition identifier on success, -1 on failure

DESCRIPTION:    Create a condition based on a value. Incoming data is evaluated with the defined value. The boolean result is returned to the expression that called the condition.

---

**setStringCondition()**


---

SYNTAX:            `conditionID = setStringCondition(data type, "element", "string", options);`

WHERE:            `data type` - type of data unit the condition applies to  
                     `element` - string, name of data unit element the condition applies to  
                     `string` - string that is used for evaluation  
                     `options` - the type of evaluation that is performed

<i>options</i>	<i>value</i>
CASE_SENSITIVE	default, evaluate case sensitive
IGNORE_CASE	ignore case
SUB_STRING	default, a match is found when only a part of a the string of an element matches the provided string
WHOLE_STRING	a match is found only when the whole string of an element matches the provided string

RETURN:            `conditionID` - condition identifier on success, -1 on failure

DESCRIPTION:     Create a condition based on a string. Incoming data is evaluated with the defined string. The boolean result is returned to the expression that called the condition.

---

**setFlagCondition()**


---

SYNTAX:            `conditionID = setFlagCondition(data type, "element", flags high, flags low);`

WHERE:            `data type` - type of data unit the condition applies to  
                     `element` - string, name of data unit element the condition applies to  
                     `flags high` - bit mask for flags that should be high for the condition to evaluate true  
                     `flags low` - bit mask for flags that should be low for the condition to evaluate true

RETURN:            `conditionID` - condition identifier on success, -1 on failure

DESCRIPTION:     Create a condition based on flag status.

For DT\_RS data units the flags of the control element can be :

RS_DTR	: DTR control line
RS_RTS	: RTS control line
RS_CTS	: CTS control line
RS_OE	: Overrun Error
RS_PE	: Parity Error
RS_FE	: Frame Error
RS_BRK	: Break Error
RS_DCD	: DCD control line
RS_RI	: RI control line
RS_DSR	: DSR control line
RS_ERR	: Any error occurred (OE, PE, FE or BRK).

---

To specify several controls for the high or low mask use the '|' sign to separate them.

---

**setTimeCondition()**


---

SYNTAX:            `conditionID = setTimeCondition(data type, "element", time, period);`

WHERE:            `data type` - type of data unit the condition applies to  
                     `element` - string, name of data unit element the condition applies to  
                     `time` - timevalue (`time.sec` and `time.usec`) with the time of day to look for as condition

---

period - timevalue with the period after the defined time the condition evaluates true

RETURN: conditionID - condition identifier on success, -1 on failure

DESCRIPTION: Create a condition based on time stamp.

---

## 2.2.2 Expressions

An expression is a logical combination of one or more conditions using Boolean syntax. They combine conditions identifiers in a single command by the use of following logical operands:

	logical OR
&&	logical AND
!	logical NOT

In addition, you can use brackets "(")" to change the order of evaluation of the operands.

### Examples

1. "condA"  
The above expression evaluates only one condition; it is true when condA is true
2. "( condA || condB ) && !condC"  
This expression is true when **either** condA **or** condB is true but **only** when condC is false (not true).
3. ""  
An empty expression is always true; this can be useful in combination with a DataTrigger component, as it will generate an event on every data unit that passes.

## 2.3 Timers

Apart from being triggered in the data stream, events they can also be generated by the timer clock of FieldCommander. This can be achieved by two components, Clock Timer and Interval Timer. A total of 100 timers can be used in a script.

### newTimer()

---

SYNTAX: timerID = newTimer();

RETURN: timerID - timer identifier on success, -1 on failure

DESCRIPTION: Create a handle for use with clock timer or interval timer

---

### stopTimer()

---

SYNTAX: stopTimer(timerID);

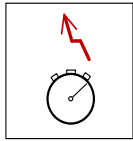
WHERE: timerID - timer identifier

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Stop the given timer.

---

### 2.3.1 Interval timer



**Name:** IntervalTimer  
**Purpose:** generate event on given timer interval  
**Input:** none  
**Output:** none

The interval timer is used to trigger events at a given interval. The interval can be as short as 10 milliseconds. This timer is often used in combination with sending data in order to poll sensors or devices.

#### startIntervalTimer()

**SYNTAX:** `startIntervalTimer(timerID, interval);`

**WHERE:** timerID - timer identifier  
interval - timevalue (time.sec and time.usec)

**RETURN:** number - 0 on success, 1 on failure

**DESCRIPTION:** Start the timer with the given interval. Every time the interval elapses, the timer generates an event until stopTimer() is called.

**EXAMPLE:**

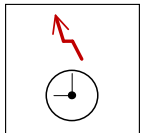
```
// set a timer at 5 seconds
time.sec = 5;
time.usec = 0;
timerID = newTimer();
startIntervalTimer(timerID, time);

sleep();

function handleEvent( event_source, event_type )
{
    switch( event_source ) {

        case timerID:
            // handle time event and restart the timer at a twice
            // as long interval
            stopTimer(timerID);
            time.sec *= 2;
            startIntervalTimer(timerID, time);
            break;
    }
}
```

### 2.3.2 Clock timer



**Name:** ClockTimer  
**Purpose:** generate event on time of day  
**Input:** none  
**Output:** none

A clock timer enables you to generate an event on a specific time of day. This works independent from the time stamps in the data stream. When you want to trap incoming data on a certain time of day, you can use a Data Trigger with a time condition. When there is no data in the stream, the DataTrigger will not generate an event. The clock timer will respect daylight saving changeovers (DST) when this function is enabled.

**startClockTimer()**

---

SYNTAX:           startClockTimer(timerID, hour, minutes [, seconds] );

WHERE:            timerID - timer identifier  
                  hour, minutes, seconds - clock time to generate event on (seconds is optional)

RETURN:           number - 0 on success, 1 on failure

DESCRIPTION:     Start the timer with the time of day. When the time occurs, a single event is generated. In case you want to stop the timer before the time occurs, call stopTimer().

EXAMPLE:          // set a clock timer at 8 AM  
                  timerID = newTimer();  
                  startClockTimer(timerID, 8, 0);  
  
                  sleep();  
  
                  function handleEvent( event\_source, event\_type )  
                  {  
                    switch( event\_source ) {  
  
                      case timerID:  
                        // handle event and set new time at 9 AM  
                        startClockTimer(timerID, 9, 0);  
                        break;  
                    }  
                  }  
                  }

---

## 2.4 Event management

### 2.4.1 Sources and types

The event mechanism makes an important factor in the way FieldCommander is used as it enables truly interactive responses from the script language. You can implement your own script code to process when a given event occurs. When a registered event occurs, the event handler `handleEvent()` is called, assuming one is available in the running script. Every event has a unique identifier and an event type, which is passed to the handler so it can be processed accordingly.

The `handleEvent()` function is called with two variables. The first variable is a source ID which identifies the component that generated the event. The second variable defines the event type that occurred.

The table below lists all available event types:

event source	event type	generated when...
Buffer	evBufferFull	the buffer is full (linear mode only)
Data trigger	evTrigger	data unit matches the defined condition(s)
Timer	evTimer	a clock or interval timer elapsed
WebGUI (hotrect)	evApplet	hotrect is clicked by an end user
FCphp	evPhp	php page calls fcTrigger()
PHP function	evPhpCompleted	exectution of php() completed
Modem port, PPP, GPRS	evModemConnected	modem connection is established
	evModemConnectFailed	modem connection could not be established
	evModemDisconnected	modem connection is terminated
Modem port	evModemRing	incoming voice call
	evModemBusy	receiving party is busy
	evModemNoDialtone	phone line is unavailable
	evModemCancelled	incoming call was cancelled (VOIP only)
TCP	evTcpConnected	TCP connection is established
	evTcpDisconnected	TCP connection is established
Converter (Modbus)	evQueueFull	command cannot be added to queue
	evTimeout	sending of command timed out
Script	evScript	FCscript calls trigger()
Email	evMail	response from mailSend() function
File transfer	evUrl	response from getUrl()/putUrl() function

## 2.4.2 Event related functions

### trigger()

SYNTAX: `trigger(source [, type]);`

WHERE: `source` - event source to generate  
`type` - (optional) event type to generate, `evScript` is default

RETURN: 0 on success, -1 on failure

DESCRIPTION: Trigger an event in the `handleEvent()` function. Be careful when calling this function from the `handleEvent()` function itself as it may result in a circular recursion, and endless loop.

### sleep()

SYNTAX: `sleep(msec);`

WHERE: `msec` - time in milliseconds that the script should be on hold. When time is omitted, the script will sleep endlessly.

RETURN: none

DESCRIPTION: Stops executing the script for a number of milliseconds. Events that occur during this time will still be handled in the script's `handleEvent` routine.

The `sleep()` function should **not** be called within from the `handleEvent()` function as this may preempt the event processing and can lead to unexpected behaviour!

Typically, a main function of a script is ended with a call to `sleep()` where the time parameter is omitted. This will force FieldCommander to "sleep" endlessly and operate in a purely event driven way. The script will "wake up" on the occurrence of an event, handle it in the `handleEvent` routine, and return to sleep mode, waiting for the next event to occur.

### waitForEvent()

SYNTAX: `waitForEvent(timeout, componentID1, componentID2...);`

WHERE: `timeout` - timevalue (`timeout.sec` and `timeout.usec`) with maximum time to wait until the event occurs.  
`componentID` - the identifier of the component that should generate an event before `waitForEvent()` continuous.

RETURN: number - 0 on success, 1 on timeout

DESCRIPTION: Suspend until one of the listed events occurs. This function should **not** be called from within the `handleEvent()` function.

EXAMPLE:

```
time.sec = 10;
time.usec = 0;
while(1) {
    waitForEvent(time, componentID1, componentID2);

    // put code here, that will be executed whenever component with
    // componentID1 or componentID2 generate an event. If no events
    // occur the code will be executed at least once in ten seconds.
}
```

### 2.4.3 Event callbacks

The code below shows a typical way to process the event callbacks through switches.

```
bufferID1 = addBuffer(sourceID, "buffer_name1", 1000, LINEAR);
bufferID2 = addBuffer(sourceID, "buffer_name2", 1000, LINEAR);
triggerID = addDataTrigger(sourceID, "", REPEAT);
timerID = newTimer();
interval.sec = 2;
interval.usec = 0;
startIntervalTimer(timerID, interval);
sleep();

function handleEvent(event_source, event_type)
{
    // handle event based on the event_source
    switch(event_source)
    {
        case bufferID1: // your code when buffer 1 is full
            break;
        case triggerID: // your code when a data unit is received
            break;
        case timerID : // your code when the 2-seconds-timer has elapsed
            break;
    }

    // handle event base on the event_type
    switch(event_type)
    {
        case evBufferFull : // your code when buffer 1 or buffer 2 is full
            break;
        case evTrigger : // your code when a data unit is received
            break;
        case evTimer: // your code when the 2-seconds-timer has elapsed
            break;
    }
}
```

## 2.5 Data exchange

### 2.5.1 Send data patterns

In many applications, you will want to send commands (requests) to the attached serial devices. The `sendString()` function allows you to send patterns (strings) of data to the local ports. Typically, this is done periodically in order to poll the status of a device or sensor.

#### **sendString()**

---

SYNTAX: `sendString(portID, string);`

WHERE: `portID` - port identifier returned by `newLocalPort()/newUSBPort()`  
`string` - text pattern to transmit

RETURN: number - 0 on success, 1 on timeout, 2 on other failure

DESCRIPTION: Transmits a string of data to a given port.

EXAMPLE: `sendString(portID, "get value of sensor 1");`

---

## 2.5.2 Send e-mail

FieldCommander allows you to send e-mail messages with a few simple calls in your script. In the call to `mailCreate()` you pass the recipient, subject line and file name of the body text. This text file can be uploaded via FTP or generated from the script itself. You can open a file and write text messages using standard Javascript functions. Optionally you may provide a second file name pointing to the body text in HTML formatting.

Next you can add recipients and attachments. All recipients and attachments after the `mailCreate()` call end up in the message. The message is transmitted with `mailSend()` which will report the delivery through an event. Note that email messages are not queued. You'll have to wait for `mailSend()` to report success or failure before creating a new message.

### mailCreate()

---

SYNTAX: `mailCreate(recipient, subject, textfile [, htmlfile]);`

WHERE: recipient - e-mail address to send mail message to  
subject - subject line of the message  
textfile - file to use as plain text body  
htmlfile - (optional) file to use as html body

RETURN: 0 on success, 1 on failure (file not found)

DESCRIPTION: Create an e-mail message. The recipient address can be given in either of two formats: simple ("name@domain.com") and descriptive ("Full Name <name@domain.com>"). The body files should exist on FieldCommander and are pointed to relative to the FTP root.

Note that email messages are not queued. You'll have to wait for `mailSend()` to report success or failure before creating a new message.

EXAMPLE: 

```
// create a plain text message
mailCreate("cer@cer.com", "For your info", "/data/fyi.txt");

// create a message with plain text + html formatting
mailCreate("CER International <cer@cer.com>", "For your info",
    "/data/fyi.txt", "/data/fyi.html");
```

### mailAddRecipient()

---

SYNTAX: `mailAddRecipient(recipient [, mode]);`

WHERE: recipient - e-mail address to send mail message to  
mode - (optional) specifies the recipient type: MAIL\_TO or MAIL\_CC

RETURN: 0 on success, 1 on failure

DESCRIPTION: Adds a recipient to an e-mail. The address can be given in either of two formats: simple ("name@domain.com") and descriptive ("Full Name <name@domain.com>"). When *mode* is omitted, the recipient is added to the "To" list.

EXAMPLE: 

```
// add address to receive the message too
mailAddRecipient("support@cer.com");

// add address to receive a carbon copy (CC) of the message
mailAddRecipient("CER Support <support@cer.com>", MAIL_CC);
```

**mailAddAttachment()**

SYNTAX: `mailAddAttachment(filename [, filetype [, subtype]]);`

WHERE: filename - file to attach to mail message  
 filetype - (optional) MIME type, possible choices:  
     MAIL\_APPLICATION (default)  
     MAIL\_TEXT  
     MAIL\_IMAGE  
     MAIL\_AUDIO  
     MAIL\_VIDEO  
 subtype - (optional) MIME subtype, default is "octet-stream"

RETURN: 0 on success, 1 on failure (file not found)

DESCRIPTION: Adds a file attachment to an e-mail. The files should exist on FieldCommander and is pointed to relative to the FTP root. By default, attachments are passed as MIME type "application/octet-stream" which should be understood by all mail clients. For more information on MIME type, check RFC2046.

EXAMPLE: `// add file to the the message  
 mailAddAttachment("/data/logdata.txt");  
  
 // add file to the the message and describe the content type  
 mailAddAttachment("/data/logdata.txt", MAIL_TEXT, "plain");`

**mailSend()**

SYNTAX: `mailSend();`

RETURN: 0 on success, 1 on failure

DESCRIPTION: Send mail message - prepared by `mailCreate()` - to the SMTP server. This function will return immediately and send the message in the background. When the function does not return a failure, that does not necessarily mean the messages was send succesfully.

The background process will report back to the script by way of an event of type **evMail**. The event source provides information on the success or problem. Note that email messages are not queued. You'll have to wait for `mailSend()` to report success or failure before creating a new message.

Common codes returned include:

event	what	resolution
0	mail successfully sent	n/a
1	internal error	n/a
2	error parsing mail	check if <code>mailCreate()</code> was success
11	error connecting to SMTP server	check SMTP and TCP/IP settings
20	e-mail address was refused	check e-mail address(es)

Note that you'll need to enable and set up the ability to send mail in the system configuration first. You must set up the SMTP server, e-mail address of the sender and possibly a gateway and DNS information first. See the *User's Guide* for details.

```

EXAMPLE:    function main() {

                mailCreate("CER International <cer@cer.com>",
                            "For your info", "/data/fyi.txt", "/data/fyi.html");
                mailAddRecipient("CER Support <support@cer.com>", MAIL_CC);
                mailAddAttachment("/data/logdata.txt", MAIL_TEXT, "plain");
                retval = mailSend();
                writeLog("mailSend: "+retval);

                sleep();
            }

            function handleEvent(event, type) {

                if (type == evMail)
                {
                    switch (event)
                    {
                        case 0: writeLog("Mail successfully sent"); break;
                        case 1: writeLog("Internal error parsing options, not sent"); break;
                        case 2: writeLog("Error parsing mail (mailCreate failed?)"); break;
                        case 11: writeLog("Error connecting to SMTP server"); break;
                        case 20: writeLog("Recipient or sender address was refused"); break;
                        default: writeLog("Email error: "+event); break;
                    }
                }
            }
        }
    }

```

### 2.5.3 Send/receive files

(in selected models)

FieldCommander features an FTP client which allows you to send and receive files from the script. The functions `putUrl()` and `getUrl()` allow you to download and upload files from any FTP and HTTP server. This can be used to store data files on remote servers, or get files from a remote FTP/HTTP server. You can upload log files to a central FTP server to save space on FieldCommander's disk for instance, or download data files to be used in your application.

The file transfer takes place in the background, the FCscript function below returns immediately so script processing continues normally. The background process will report back to the script by way of an event of type **evUrl**. The event source provides information on the success or problem. Note that the commands are not queued. You'll have to wait for a response before starting a new transfer.

event	what	resolution
0	transfer successful	n/a
1	protocol in URL is not supported	only http, https, ftp and ftps work
3	the URL was not properly formatted	check URL
6	given remote host could not be resolved	check URL and DNS settings
7	failed to connect to remote host	check URL and FTP server status
8	remote host is not at an OK FTP server	FTP server is incompatible
9	access had been denied by FTP server	check account settings on FTP

---

**putUrl()** **in selected models**

---

SYNTAX: `putUrl(url, file, user, password, transfermode, filemode);`

WHERE: `url` - full location of the target file  
`file` - full location of the source file to transmit (path + filename)  
`user` - user name for login authentication  
`password` - password for login authentication  
`transfermode` - mode for data transfer: TM\_ASCII | TM\_BINARY  
`filemode` - mode for existing files: FM\_CLEAR | FM\_APPEND

RETURN: 0 on success, an event of type **evUrl** will return the status of the transfer  
1 on failure, the transfer could not be started, check the arguments

DESCRIPTION: Transmit ("upload") a file. `putUrl()` sends a *file* to an `ftp://` location given by the *url*. When the target host asks for authentication, the *user* and *password* is provided. The *transfermode* specifies whether the source file is ASCII (plain text) or binary.

*Filemode* specifies what to do when the target file already exists: FM\_CLEAR overwrites the file, FM\_APPEND adds the source file content to the existing file. FM\_APPEND is typically only meaningful when transferring ASCII text files.

Note that you need write permissions in the target directory. To work with domain names instead of IP numbers, you should enable DNS in the System Manager (see User's Guide)

EXAMPLE: `putUrl("ftp://ftp.myhost.com/pub/mymessage.txt",  
"/data/mymessage.txt", "guest", "guest", TM_ASCII, FM_CLEAR);`

---

---

**getUrl()** **in selected models**

---

SYNTAX: `getUrl(url, file, user, password, transfermode);`

WHERE: `url` - full location of the source file to receive  
`file` - full location of the target file (path +filename)  
`user` - user name for login authentication  
`password` - password for login authentication  
`transfermode` - mode for data transfer: TM\_ASCII | TM\_BINARY

RETURN: 0 on success, an event of type **evUrl** will return the status of the transfer  
1 on failure, the transfer could not be started, check the arguments

DESCRIPTION: Receive ("download") a file. `getUrl()` retrieves a file from a `http://` or `ftp://` location given by *url*. When the source host asks for authentication, the *user* and *password* is provided. The *transfermode* specifies whether the source file is ASCII (plain text) or binary.

To work with domain names instead of IP numbers, you must enable DNS in the System Manager (see User's Guide)

EXAMPLE: `getUrl("ftp://ftp.host.com/pub/this.txt",  
"/data/target.txt", "guest", "guest", TM_ASCII);`

---

## 2.5.4 Export as XML

(in selected models)

The Extensible Markup Language (XML) is the universal format for structured documents and data on the Web. XML is a set of rules (you may also think of them as guidelines or conventions) for designing text formats that let you structure your data. Like HTML, XML makes use of *tags* (words bracketed by '<' and '>') and *attributes* (of the form name="value"). While HTML specifies what each tag and attribute means, and often how the text between them will look in a browser, XML uses the tags only to delimit pieces of data, and leaves the interpretation of the data completely to the application that reads it.

FieldCommander can send your structured XML documents over a TCP socket stream to a 3<sup>rd</sup> party device or application which reads it. Analogue to the web pages, FieldCommander will parse XML templates and replace the meta-tags with up-to-date values. It uses the same meta-table as discussed in the next paragraph.

Your XML template, uploaded to the /DATA directory, may look like this:

```
<XML>
  <logentry>
    <room><!--#meta R20ROOMNAME --></room>
    <temperature><!--#meta R20TEMPERATURE --></temperature>
    <humidity><!--#meta R20HUMIDITY --></humidity>
  </logentry>
  <logentry>
    <room><!--#meta R21ROOMNAME --></room>
    <temperature><!--#meta R21TEMPERATURE --></temperature>
    <humidity><!--#meta R21HUMIDITY --></humidity>
  </logentry>
</XML>
```

The structure of your XML template totally depends on what the receiver of the XML documents expects. You can upload your XML template to the data directory with an FTP client of your choice.

The actual transfer of the XML document is started with the script function *exportXml()*. Before exporting the XML document, all meta-tags are replaced with the current values as they are listed in the meta-table.

### **exportXml()**

**in selected models**

---

SYNTAX:	<code>exportXml(host, tcpport, file);</code>
WHERE:	host - the IP address of the remote host to export to tcpport - the TCP port of the remote host to export to file - the XML template file to export
RETURN:	number - 0 on success, 1 on failure
DESCRIPTION:	Sends an XML document based on the given template.

---

## 2.6 Meta tags

Meta tags are used to share information between your application script and user interface. As they are kept 'in memory', the tags cannot be used for permanent storage. You can define our own tags but FieldCommander provides information through predefined system tags as well, see the table below for built-in tags:

meta tag name	description
_IPADDRESS	IP address of this FieldCommander, nnn.nnn.nnn.nnn
_HOSTNAME	hostname assigned to this FieldCommander
_TIME	local system time, HH:mm:ss (24 hour clock)
_DATE	current date, mm/dd/yyyy
_VERSION	version of the software (major.minor.patchlevel)
_SERIALNUMBER	serial number of this FieldCommander
_MODEL	model type of this FieldCommander
_LOAD	CPU load in percent, average of last minute
_LOAD5	CPU load in percent, average of last 5 minutes
_LOAD15	CPU load in percent, average of last 15 minutes
_UPTIME	system uptime, formatted as text
_UPTIMESEC	system uptime in seconds
_MEMFREE	available system memory in kilo bytes (kB)
_MEMUSED	allocated system memory in kilo bytes (kB)
_DISKFREE	available disk space in kilo bytes (kB)
_DISKUSED	allocated disk space in kilo bytes (kB)
_RS232_PORTS	number of RS232 ports in the system
_RS485_PORTS	number of RS485 ports in the system
_USB_PORTS	number of USB ports in the system
_NDU<bufname>	number of data units in buffer <bufname> E.g.: "_NDUmybuffer" for buffer with name "mybuffer"

### **getMetaValue()**

SYNTAX:            value = getMetaValue(tag name);

WHERE:             tag name - the key identifying the location in the meta table

RETURN:            value - the current value of a meta tag in the meta table; when the tag is not found, an empty string is returned.

DESCRIPTION:      Reads a value from the meta table. Note that the name is case sensitive.

**setMetaValue()**

---

SYNTAX:            setMetaValue(meta tag name, value);

WHERE:             tag name - the key identifying the location in the meta table  
                    value - the content to assign to the location

RETURN:            number - 0 on success, 1 on failure

DESCRIPTION:      Writes a value to the meta table. When the *meta tag name* was not used before, a new entry is added to the table. System meta tags cannot be set.

Notes:

- *meta tag name* is case sensitive
  - *meta tag name* is can contain a number or texts up to 64 characters
  - *meta tag value* is can contain a number or texts up to 256 characters
- 

**clearMetaValue()**

---

SYNTAX:            clearMetaValue(tag name);

WHERE:             meta tag name - the key identifying the location in the meta table

RETURN:            number - 0 on success, 1 on failure

DESCRIPTION:      Delete a value from the meta table. System meta tags cannot be removed.

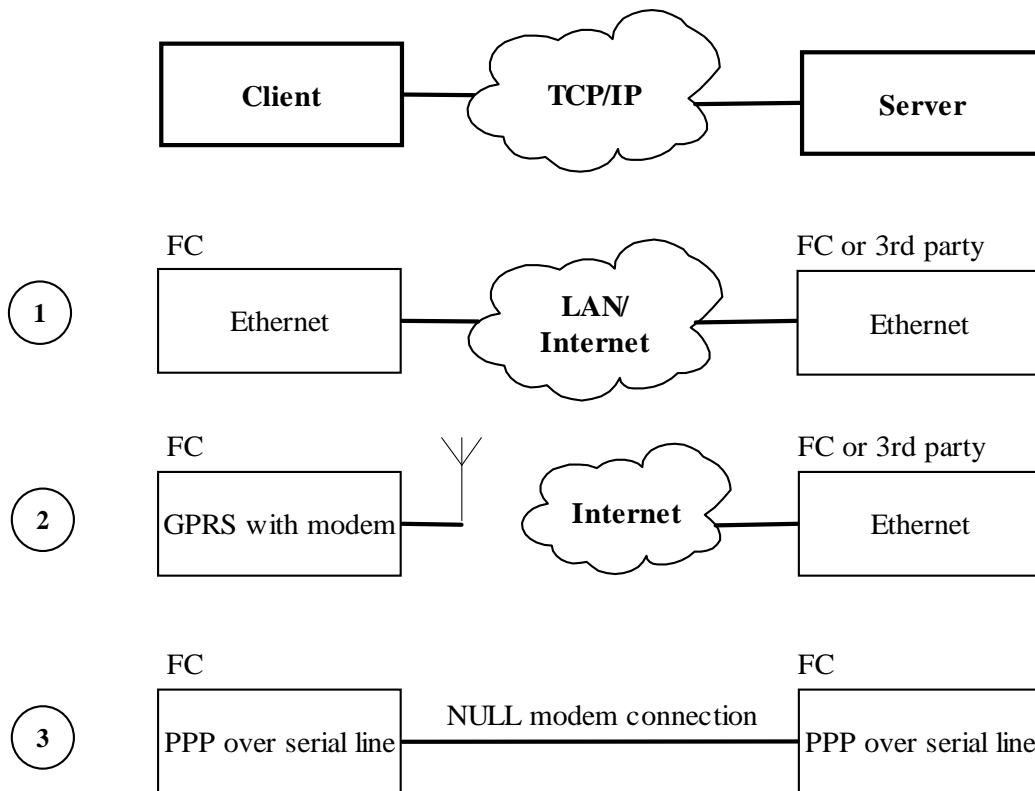
---

## 2.7 TCP communication

FieldCommander provides several ways to communicate over TCP/IP networks. You can send data units or text strings between FieldCommander units, or 3<sup>rd</sup> party applicaitons. And TCP/IP is not restricted to the Ethernet connection alone. GPRS and PPP allow you to employ it even where you don't have a fixed network.

### 2.7.1 Setting up the link

Although TCP/IP is widely known as the the protocols to drive Internet and many networks, they are not limited to the Ethernet network interface. FieldCommander can employ several paths for TCP/IP connections.



**1)** This is the "conventional" approach where FieldCommander uses its Ethernet port for TCP/IP communication. This interface is always available and you don't have to do anything in the script to set it up. The next paragraph explains the possible ways to communicate over the connection.

**2)** In many remote and mobile applications, Ethernet is not available. GPRS, a data packet service available from mobile phone providers, can serve as an alternative medium. When FieldCommander is equipped with a GPRS capable GSM modem, it can set up a connection to Internet through the network of your mobile phone service provider.

You'll have to tell the client how to set up the GPRS connection and to what IP address you want to send data to:

*Client:*

```

mpl = newModemPort(1, MODEM_GSM_GPRS);
setLocalPort(mpl, MODEM_PIN, "0000");
setLocalPort(mpl, MODEM_APN, "live.vodafone.com");
setLocalPort(mpl, MODEM_USER, "vodafone");
setLocalPort(mpl, MODEM_PASSWORD, "vodafone");
openPorts();
gprsConnect(mpl, "82.120.100.100");

```

The server should have access to the internet though the Ethernet interface, you don't have to do anything extra in the script set it up. The next paragraph explains the possible ways to communicate over the connection.

**3)** An alternative way to set up a TCP connection is over a serial line. The physical connection can be a NULL modem cable but can also be established over a leased line. This latter requires leased line modems, forming a secure, cost effective and reliable connection method.

To establish a TCP link between the two serial ports, you only have to make a reservation for the port using `newModemPort()` and call `pppConnect()`. Both sides use the same lines in the script as the point-to-point connection works in both directions.

*Client:*

```
mp1 = newModemPort(1, MODEM_NULL);
openPorts();
pppConnect(p1);
```

*Server:*

```
mp1 = newModemPort(1, MODEM_NULL);
openPorts();
pppConnect(mp1);
```

### **gprsConnect()**

---

SYNTAX: `connID = gprsConnect(portID, "address");`

WHERE: `portID` - port identifier returned by `newLocalPort()`  
`address` - the IP address of the target FieldCommander unit (host)

RETURN: connection identifier on success, -1 on failure

DESCRIPTION: Establish GPRS connection. This function returns immediately. When it returns successfully, that does not mean that connection is already there. An event of type **evModemConnected** will be triggered when the connection is (re)established. In case the connection could not be established, **evModemConnectFailed** is issued.

---

### **gprsDisconnect()**

---

SYNTAX: `gprsDisconnect(connID);`

WHERE: `connID` - the connection identifier returned by `gprsConnect()`

RETURN: number - 0 on success, -1 on failure

DESCRIPTION: Terminate an GPRS connection. This function returns immediately. When it returns successfully, that does not mean that connection is already gone. An event of type **evModemDisconnected** will be triggered when the connection is terminated.

---

### **pppConnect()**

---

SYNTAX: `connID = pppConnect(portID);`

WHERE: `portID` - port identifier returned by `newLocalPort()`

RETURN: connection identifier on success, -1 on failure

DESCRIPTION: Establish PPP connection. This function returns immediately. When it returns successfully, that does not mean that connection is already there. An event of type **evModemConnected** will be triggered when the connection is (re)established. In case the connection could not be established, **evModemConnectFailed** is issued.

---

**pppDisconnect()**

SYNTAX: `pppDisconnect(connID);`

WHERE: `connID` - the connection identifier returned by `pppConnect()`

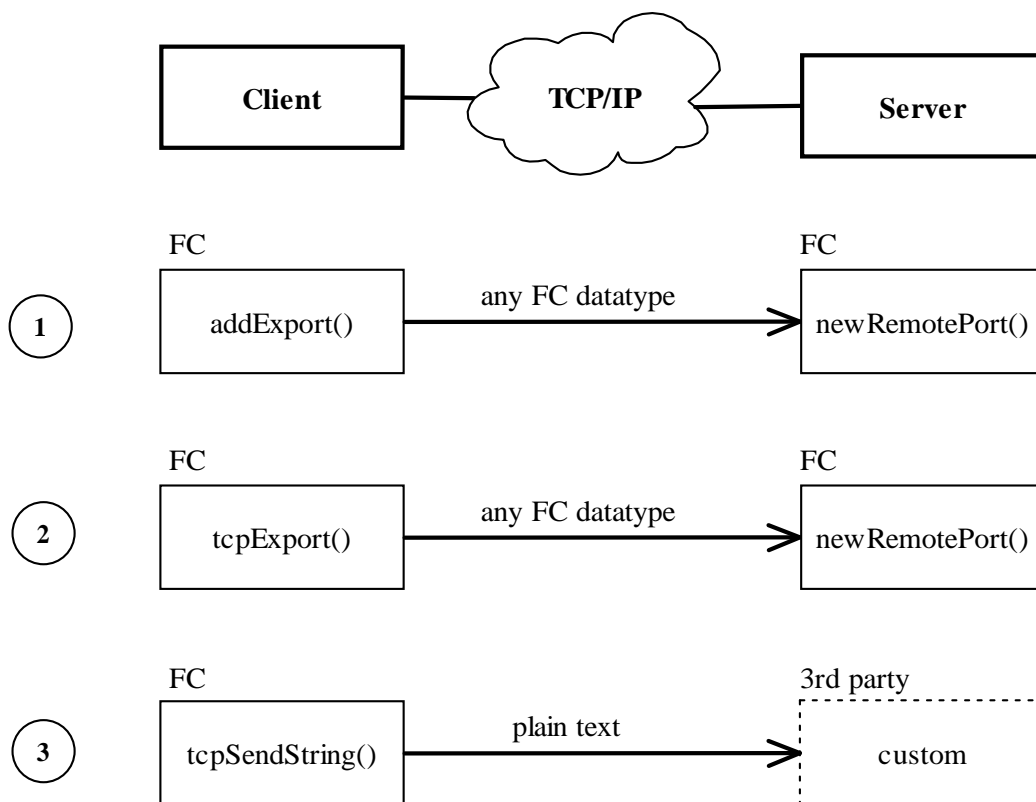
RETURN: number - 0 on success, -1 on failure

DESCRIPTION: Terminate an PPP connection. This function returns immediately. When it returns successfully, that does not mean that connection is already gone. An event of type **evModemDisconnected** will be triggered when the connection is terminated.

**2.7.2 Establish a connection****2.7.3 Sending data**

FieldCommander provides several ways to communicate over TCP/IP networks. Besides for networking services such as HTTP, FTP, SMTP, DHCP etc, it is possible to transfer data units or even plain text through a TCP/IP interface.

The picture below shows three communication methods. Each of them assume a "client" to produce the data, and a "server" to consume/receive it. They all use plain TCP socket communication; the server opens a port and listens until a client connects and sends data. After a transfer, the client may terminate the connection.



**1)** provides a convenient way to send all data units in a stream over TCP/IP to a receiving FieldCommander unit. It may - for example - export all data units (DT\_RS) it acquires on its local port 1. The server receives the dataunits in the same fashion as if they came from one of its own local ports. These example use TCP port 1100.

*Client (IP=192.168.10.100):*

```
lp1 = addLocalPort(1, DT_RS);
addExport(lp1, "192.168.10.200", 1100);
```

*Server (IP=192.168.10.200):*

```
rp1 = newRemotePort(1100, DT_RS);
```

**2)** works similar as seen from the server, but instead of passing all units in the stream, the client now exports a single data unit at a time by calling `tcpExport()` in the `handleEvent()` function of FCscript. The connection should be established first by calling `tcpConnect()`.

*Client (IP=192.168.10.100):*

```
sock1 = tcpConnect("192.168.10.200", 1100);
// assume a buffer with ASCII frames is set up as bu1
du = getBufferDataUnit(bu1, DT_AF)
tcpExport(sock1, DT_AF, du);
```

*Server (IP=192.168.10.200):*

```
rp1 = newRemotePort(1100, DT_AF);
```

**3)** uses the same strategy as in method 2, but instead of sending a data unit (whose format is FieldCommander proprietary), it transmits plain text strings over the socket connection. This allows you to build your own application to receive and process information send from FieldCommander.

*Client (IP=192.168.10.100):*

```
sock1 = tcpConnect("192.168.10.202", 1100);
tcpSendString(sock1, "Put some text here");
```

*Server (IP=192.168.10.202):*

The receiving application is up to you. In terms of TCP/IP, it should be noted that the socket uses a IPv4 based stream socket. To get ready to accept a connection, the following steps should be performed (with their common function call in *italic*)

- create socket: *socket()*;
- bind to IP address and port number: *bind()*;
- prepare for connections: *listen()*;
- wait for incoming connections: *accept()*;
- when connection is accepted, just *read()* the packets until client disconnects.

**newRemotePort()**


---

SYNTAX: `portID = newRemotePort(portnumber, datatype);`

WHERE: `portnumber` - TCP port number that is used for remote connections  
`datatype` - the data unit type that the imported stream contains

RETURN: `portID` - port identifier on success, -1 on failure

DESCRIPTION: Opens a Remote Port. Other FieldCommander devices can connect over the TCP/IP network to the Remote Port with `addExport()` or `tcpConnect()/tcpExport()`.

---

**addExport()**


---

SYNTAX: `exportID = addExport(sourceID, "address", port number);`

WHERE: `sourceID` - the source component identifier  
`address` - the IP address of the target FieldCommander unit  
`port number` - TCP/IP port number that is used to contact the target FieldCommander unit.

RETURN: `exportID` - export identifier on success, -1 on failure

DESCRIPTION: Opens a TCP socket connection to a Remote Port on another FieldCommander. It exports the stream of data units to the remote FieldCommander unit.

---

**tcpConnect()**


---

SYNTAX: `socketID = tcpConnect("address", port number);`

WHERE: `address` - the IP address of the target FieldCommander unit  
`port number` - TCP/IP port number that is used to contact the target FieldCommander unit.

RETURN: `socketID` - socket identifier on success, -1 on failure

DESCRIPTION: Opens a TCP socket connection to a Remote Port on another FieldCommander or a 3<sup>rd</sup> party application listening on the specified TCP port. To export data units to another FieldCommander, call the `tcpExport()` function. For connections with 3<sup>rd</sup> party applications, use `tcpSendString()`.

This function returns immediately. When it returns success, that doesn't mean the connection is already there. An event of type **evTcpConnected** is triggered when the connection is established. When the connection terminates for whatever reason, an event with type **evTcpDisconnected** is triggered and you should call `tcpConnect()` again before sending data.

---

**tcpExport()**


---

SYNTAX: `tcpExport(socketID, datatype, data object);`

WHERE: `socketID` - the socket identifier returned by `tcpConnect()`  
`datatype` - the data unit type that the imported stream contains, such as `DT_RS`  
`data object` - a data unit object filled according to the given `datatype`

RETURN: `number` - 0 on success, 1 on failure

DESCRIPTION: Sends a data unit over the TCP socket connection to a Remote Port on another FieldCommander.

---

**tcpSendString()**

SYNTAX: `tcpSendString(socketID, text);`

WHERE: `socketID` - the socket identifier returned by `tcpConnect()`  
`text` - the text string to send

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Sends a plain text over the TCP socket connection to a 3<sup>rd</sup>-party application.

**tcpDisconnect()**

SYNTAX: `tcpDisconnect(socketID);`

WHERE: `socketID` - the socket identifier returned by `tcpConnect()`

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Terminate an open TCP connection. This function returns immediately. When it returns success, that doesn't mean the connection is already gone. An event of type **evTcpDisconnected** is triggered when the connection is terminated.

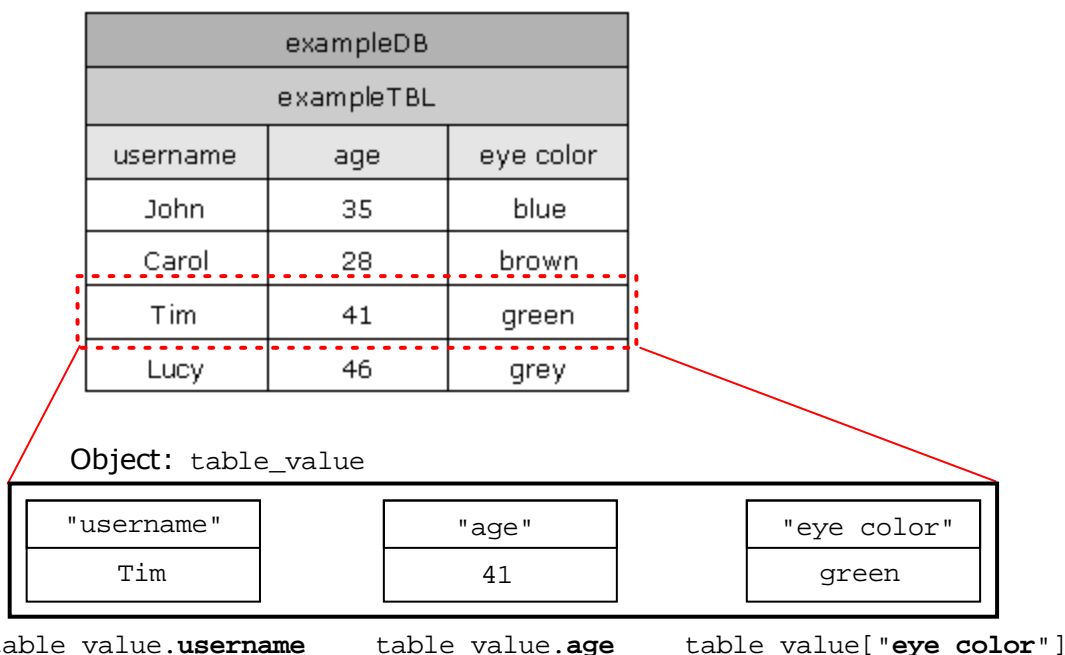
## 2.8 Database access

FieldCommander features a powerful database engine used to store structured information and share it between your application in FCscript and user interface based on FCphp.

### 2.8.1 Access from script

FieldCommander supports two ways to use the database. Depending on your needs or experience you can use either the set of simple functions, or execute true SQL queries. Note that you can only open existing databases and tables from the script, not create them. Databases and tables can be created using the appropriate FCphp functions or with the web interface.

Consider the example table below:



Database "exampleDB" contains a single table called "exampleTBL". Each column in a table uses a unique, case sensitive name; in this case there are 3 columns named "username", "age" and "eye color". There are 4 rows with data, with each row containing 3 separate entries called data fields. The values in the data fields of the first column are used to index the row they are in. Data is requested from a table one row at a time.

In our example, the first column "username" contains a list of names. To select and use data from this table, FieldCommander uses several script commands. In the following example, we will request the value "age" for "username" Tim.

First of all, you must tell FieldCommander which database to look for with the `setConfigDatabase()` command. In this case the command would be:

```
setConfigDatabase("exampleDB");
```

When the database is found, you can use the `getConfigTableRow()` command to retrieve information from a row and assign it to an object : in this case the object called *table\_value*. You must provide both table name and index value to select the proper row.

```
table_value = getConfigTableRow("exampleTBL", "Tim");
```

Upon each request, FieldCommander checks whether a row with that index value exists, and if so, returns an object. That object contains object members: the value of each data field in a row is assigned to an object member with the same name label as the column the data field is in. The value of each separate object member can be assigned to a script variable. In our example, the object *table\_value* contains all data from row with index value "Tim". You can use one of two script commands to assign the value of the object member to a variable:

```
a_persons_age = table_value.age;  
a_persons_age = table_value["age"];
```

In a similar way, you can also assign the value of object members "eye color" or "username" to a script variable. Note that because of the white space in column name "eye color" you **must** use:

```
a_persons_eyecolor = table_value["eye color"];
```

rather than:

```
a_persons_eyecolor = table_value.eye color;
```

## 2.8.2 Access the database

The list below has simple functions to set and get information in the tables:

function	description
<code>setConfigDatabase()</code>	Set database to use by the following commands
<code>getConfigTableRow()</code>	Retrieve data from a specific row indicated by its index value and return an object containing the rows data
<code>getFirstConfigTableRow()</code>	Retrieve data in the first row of a table and return an object containing the rows data
<code>getNextConfigTableRow()</code>	Retrieve data in the next row of the result set and return an object containing the row's data
<code>getLastConfigTableRow()</code>	Retrieve data in the last row of a table and return an object containing the row's data
<code>getPreviousConfigTableRow()</code>	Retrieve data in the previous row in the result set and return an object containing the row's data
<code>setConfigTableRow()</code>	Add a row or change data in a row of a table
<code>deleteConfigTableRow()</code>	Delete a row from a table

You can also your own SQL queries to access tables if you please, check 2.7.3 for the SQL functions.

### **setConfigDatabase()**

SYNTAX: `setConfigDatabase(db_name);`

WHERE: `db_name` - the name of the database file without the \*.db extension that is situated in the /database directory and that should be referred to as the configuration database.

RETURN: 0 on success, >0 on failure (database file does not exist)

DESCRIPTION: Define which database should be used by the script and open the database. Only one database can be assigned to the script at each moment and if the command is called again, the previous assigned database is closed and the new one is opened.

### **getConfigTableRow()**

SYNTAX: `object = getConfigTableRow(table_name, index_value);`

WHERE: `table_name` - name of the table were the data has to be fetch from.  
`index_value` - value of the index were the data is fetched from. The first field of the table is always the index value and should be unique.

RETURN: `object` - object containing all data from the specified index row. Data from each data field (including the index) is stored in a separate object member with the same name label as the column the data field is in. If the specified index value was not found, or there is no such table in the database, null is returned.

DESCRIPTION: Request all data in a single row of a specific table and return an object containing all data.

---

```

EXAMPLE:    //
            // example 1 - get
            //

            setConfigDatabase("exampleDB");

            if ( (userdata=getConfigTableRow("exampleTBL", "John")) != null ) {
                writeLog("username: "+userdata.username+", age: "+userdata.age+", eye
                color: "+userdata["eye color"]);
            }

            // output

            username: John, age: 35, eye color: blue

```

---

### **getFirstConfigTableRow()**

---

**SYNTAX:**        object = getFirstConfigTableRow(table\_name);

**WHERE:**         table\_name - name of the table were the data has to be fetch from

**RETURN:**        object - object containing the data from the first row. If the specified row was not found, or there is no such table in the database null is returned.

**DESCRIPTION:**   Request all data in the first row of the selected table and return an object containing all data.

---

### **getNextConfigTableRow()**

---

**SYNTAX:**        object = getNextConfigTableRow(table\_name);

**WHERE:**         table\_name - name of the table were the data has to be fetch from.

**RETURN:**        object - object containing the data from the next row. If the specified row was not found, or there is no such table in the database null is returned.

**DESCRIPTION:**   Request all data in the next row relative to the current position in the current table and return an object containing all data. Use either getConfigtableRow(), getPreviousConfigTableRow() or getFirstConfigTableRow() to determine an initial position in a table.

```

EXAMPLE:    //
            // example 2 - first / next
            //

            setConfigDatabase("exampleDB");

            userdata=getFirstConfigTableRow("exampleTBL");

            while ( userdata != null ) {
                writeLog("username: "+userdata.username+", age: "
                +userdata.age+", eye color: "+userdata["eye color"]);
                userdata=getNextConfigTableRow("exampleTBL");
            }

            // output

            username: John, age: 35, eye color: blue
            username: Carol, age: 28, eye color: brown
            username: Tim, age: 41, eye color: green
            username: Lucy, age: 46, eye color: grey

```

---

**getLastConfigTableRow()**


---

SYNTAX:	<code>object = getLastConfigTableRow(table_name);</code>
WHERE:	<code>table_name</code> - name of the table were the data has to be fetch from.
RETURN:	<code>object</code> - object containing all data from the last row. If the specified row was not found, or there is no such table in the database null is returned.
DESCRIPTION:	Request all data in the last row of the selected table and return an object containing all data.

---

**getPreviousConfigTableRow()**


---

SYNTAX:	<code>object = getPreviousConfigTableRow(table_name);</code>
WHERE:	<code>table name</code> - name of the table were the data has to be fetch from
RETURN:	<code>object</code> - object containing the data from the previous row. If the specified row was not found, or there is no such table in the database null is returned
DESCRIPTION:	Request all data in the previous row relative to the current position in the current table and return an object containing all data. Use either <code>getConfigTableRow()</code> , <code>getPreviousConfigTableRow()</code> or <code>getLastConfigTableRow()</code> to determine an initial position in a table.
EXAMPLE:	<pre> setConfigDatabase("exampleDB"); userdata=getLastConfigTableRow("exampleTBL");  while ( userdata != null ) {     writeLog("username: "+userdata.username+", age: "         +userdata.age+", eye color: "+userdata["eye color"]);     userdata=getPreviousConfigTableRow("exampleTBL"); }  // output  username: Lucy, age: 46, eye color: grey username: Tim, age: 41, eye color: green username: Carol, age: 28, eye color: brown username: John, age: 35, eye color: blue </pre>

---

**setConfigTableRow()**


---

SYNTAX:	<code>setConfigTableRow(table_name, object [, replacemode]);</code>
WHERE:	<code>table_name</code> - name of the table were the data has to be stored in <code>object</code> - object containing the data <code>replacemode</code> - (optional) when <code>TRUE</code> (default), existing rows are replaced; when <code>FALSE</code> , the this function returns failure in case the index already exists.
RETURN:	0 on success, >0 on failure.
DESCRIPTION:	Write data to a row in the selected table.  If the index value (first column) does not match a row that is already in the table, a new row is added. When the row is in the table it will be overwritten, unless <code>replacemode</code> is set of <code>FALSE</code> . Table fields that have no matching member in the object are filled with <code>NULL</code> .
EXAMPLE:	<code>setConfigDatabase("exampleDB");</code>

---

```
// Add a new username to the exampleTBL table

var userdata = new Object();
userdata["username"] = "Harry";
userdata["age"]      = 20;
userdata["eye color"] = "yellow";

if ( setConfigTableRow("exampleTBL", userdata) != 0 ) {
    writeLog("Adding of table row failed");
}

// Change the age of a username in the exampleTBL table

var userdata = new Object();
userdata.username = "John";
userdata.age      = "37";

if ( setConfigTableRow("exampleTBL", userdata, true ) != 0 ) {
    writeLog("Editing of table row failed");
}
```

---

### **deleteConfigTableRow()**

---

SYNTAX: `deleteConfigTableRow(table_name, index_value);`

WHERE: `table_name` - name of the table were the data has to be deleted from  
`index_value` - value of the index were the row of data has to be deleted from. If the index value or table does not exist, the function returns failure.

RETURN: 0 on success, >0 on failure.

DESCRIPTION: Delete the selected row from the selected table.

EXAMPLE:

```
//
// example 5 - delete
//

setConfigDatabase("exampleDB");

if ( deleteConfigTableRow("exampleTBL", "Harry") != 0 ) {
    writeLog("Deleting the table row failed");
}
```

---

### 2.8.3 Access by SQL queries

When you need more flexibility than the previous table function provide, you can access the database using SQL queries. Refer to the *User's Guide* for information regarding to the supported SQL commands and syntax.

function	description
<code>setConfigDatabase()</code>	Set database to use by the following commands
<code>configTableQuery()</code>	Execute a true SQL query
<code>getConfigTableRow()</code>	Get (next) row from the SQL query result set
<code>configTableQueryFinalize()</code>	Free the current result set manually

#### **setConfigDatabase()**

SYNTAX: `setConfigDatabase(db_name);`

WHERE: `db_name` - the name of the database file without the \*.db extension that is situated in the /database directory and that should be referred to as the configuration database.

RETURN: 0 on success, >0 on failure (database file does not exist)

DESCRIPTION: Define which database should be used by the script and open the database. Only one database can be assigned to the script at each moment and if the command is called again, the previous assigned database is closed and the new one is opened.

#### **configTableQuery()**

SYNTAX: `configTableQuery(sql_query);`

WHERE: `sql_query` - string with the SQL query

RETURN: 0 on success, >0 on failure

DESCRIPTION: Execute an SQL query. When the query is supposed to return results, these are fetched by calling `getConfigTableRow()`. The SQL syntax is listed the *User's Guide*.

#### **getConfigTableRow()**

SYNTAX: `getConfigTableQeuryRow();`

RETURN: object on success or null on when no (more) results

DESCRIPTION: Get an object with column names and values of next row in result set from query.

EXAMPLE:

```
// dump Users table from database Demo to log file
setConfigDatabase("Demo");
configTableQuery("SELECT * FROM Users");
while (row = getConfigTableRow() ) {
    line = "";
    for (var cell in row)
        line += row[cell]+'\\t';
    writeLog(line);
}
```

```
}
```

---

**configTableQueryFinalize()**

---

SYNTAX:            `configTableQueryFinalize();`

RETURN:            0 on success, >0 on failure

DESCRIPTION:      Free the memory occupied by the existing result set. This function is automatically called before `configTableQuery()` so normally it is not necessary to call it by yourself unless you see "*Database is locked*" messages.

---

## 2.9 Modem functions

FieldCommander can communicate with the outside world through -optional- external modems.

The modem should be properly wired and set up before FieldCommander can use it. A modem will typically be connected to a RS-232 port with a straight 9-pin cable. GSM modems require a SIM card to operate. See `newLocalPort()` and `setLocalPort()` for details on how to set up the modem port.

### 2.9.1 Call setup

The call setup is approached through a combination of modem functions and event callbacks. The functions `modemDial()`, `modemAnswer()` and `modemHangup()` perform the actions where the respective changes in the modem state are reported through the following events:

event type	description
<code>evModemConnected</code>	connection is established
<code>evModemConnectFailed</code>	connection could not be established
<code>evModemDisconnected</code>	connection is terminated
<code>evModemRing</code>	incoming voice call
<code>evModemBusy</code>	receiving party is busy
<code>evModemNoDialtone</code>	phone line is unavailable

#### **modemDial()**

---

SYNTAX: `modemDial(portID, phone number);`

WHERE: `portID` - port identifier returned by `newModemPort()`  
`phone number` - phone number (international format) to call

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Set up a voice call with the given phone number. The function returns immediately and will report the calling state through events. See table above.

Depending on your contract, you will probably be charged for every call you make.

EXAMPLE: `modemDial(portID, "+31612345678");`

---

#### **modemAnswer()**

---

SYNTAX: `modemAnswer(portID);`

WHERE: `portID` - port identifier returned by `newModemPort()`

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Answer an incoming voice call. The function returns immediately and will report the calling state through events. See table above.

Depending on your contract, you may be charged for calls you answer.

---

```

EXAMPLE:    // Example switch case in handleEvent(),
            // accept call from +31612345678

            case evModemRing:
                phoneId = modemGetCaller(portID);
                if (phoneId == "+31612345678")
                    modemAnswer(portID);
                else
                    modemHangup(portID);
                break;

```

---

### **modemHangup()**

---

SYNTAX: `modemHangup(portID);`

WHERE: `portID` - port identifier returned by `newModemPort()`

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Terminate an ongoing voice call. The function returns immediately and will report the calling state through events. See table above.

EXAMPLE: `modemHangup(portID);`

---

### **modemGetCaller()**

---

SYNTAX: `modemDial(portID);`

WHERE: `portID` - port identifier returned by `newModemPort()`

RETURN: string - phone number (international format) on success, empty string otherwise

DESCRIPTION: Get the phone number (caller line identification) of the calling party. When the number could not be determined, an empty string is returned.

EXAMPLE: `phoneId = modemGetCaller(portID);`  
`if (phoneId != "")`  
 `writeLog(phoneId+"is calling!");`  
`else`  
 `writeLog("Someone is calling!");`

---

## **2.9.2 Send text message**

When you hook up the (optional) modem for cellular communications, incoming SMS text messages are automatically retrieved from a data stream in FieldCommander once you opened the port appropriately (see `newLocalPort()`). The same modem can be used to send out a text message with a single call to the script function `sendSMS()`. The modem should support the ASCII based AT command set, defined in the ETSI GSM 07.05 protocol.

### **sendSMS()**

---

SYNTAX: `sendSMS(portID, message, recipient phone);`

WHERE: `portID` - port identifier returned by `newModemPort()`  
`message` - actual text message to send (max 160 char)  
`recipient phone` - phone number (international format to send text message to)

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Send an SMS text message through the external GSM modem (optional).

---

The port must be opened using `newModemPort()`.

The modem should be powered and a valid SIM card must be inserted. The phone number of the providers SMS message center and SIM cards PIN code should correctly be set through `setLocalPort()`. Depending on your mobile contract, you will probably be charged for every SMS message you send.

```
EXAMPLE:  portID = newModemPort(1, MODEM_GSM);
          setLocalPort(portID, MODEM_PIN, "0000");
          setLocalPort(portID, MODEM_SMSCENTER, "+31612345000");
          sendSMS(portID, "Warning: level in tank A is above 31.2", "+31612345678");
```

---

### 2.9.3 Send multimedia message

(in selected models)

The cellular modem (optional), cannot only send short text messages but also pictures. This is called multimedia messaging or MMS. With the `sendMMS()` function you can pass a picture along with a text message and subject line to one or more recipients. A recipient can be either the number of a mobile phone an e-mail address. The message is delivered through your operator.

#### **sendMMS()**

**in selected models**

SYNTAX: `sendMMS(portID, filename, subject, message, recipient [, recipient]);`

WHERE: `portID` - port identifier returned by `newModemPort()`  
`filename` - name of the image to send (.jpg or .gif)  
`subject` - actual text message to send (MAX 160 CHAR)  
`message` - text message to go along with the picture  
`recipient` - phone number (international format) or e-mail address to MMS message to;  
you can add multiple recipients by adding extra arguments)

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Send an MMS message through the external GSM modem (optional). The port must be opened using `newModemPort()`.

The modem must be powered and a valid SIM card must be inserted. Refer to `setLocalPort()` to see which settings must be configured before calling this function. Depending on your mobile contract, you will probably be charged for every recipient of each MMS message you send.

```
EXAMPLE:  sendMMS(portID, "/www/tmp/photo.jpg", "Our picture", "", "+31612345678");
```

---

## 2.10 Audio functions

(in selected models)

FieldCommander can use the built-in audio interface (available on selective models) to play audio and detect DTMF tones. With this, you can create interactive voice response applications.

Refer to `newLocalPort()` and `setLocalPort()` for details on how to set up and configure the audio port, channels and volume levels.

### 2.10.1 Play sound

#### **audioSoundPlay()**

**in selected models**

SYNTAX: `audioSoundPlay(portID, filename [, queue mode]);`

WHERE: `portID` - port identifier returned by `newAudioPort()`  
`filename` - name of the audio file to send (.wav)  
`queue mode` - (optional) `AUDIO_QUEUE` (default) or `AUDIO_DIRECT`

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Plays a sound clip through the audio interface. When sound is currently playing, the clip is added to the play queue, unless `AUDIO_DIRECT` is given as mode. In that case, the queue is emptied before the sound clip is played.

The audio file must be encoded as follows:

file format	WAV
sample rate	22050 Hz
sample size	16 bit
channels	mono
encoding	PCM encoding

EXAMPLE: 

```
// play sound
audioSoundPlay(portID, "/data/welcome.wav");

// play sound immediately (empties queue before playing)
audioSoundPlay(portID, "/data/goodbye.wav", AUDIO_DIRECT);
```

#### **audioSoundStop()**

**in selected models**

SYNTAX: `audioSoundStop(portID);`

WHERE: `portID` - port identifier returned by `newAudioPort()`

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Interrupts the audio currently playing and emptied the queue.

EXAMPLE: `audioSoundStop(portID);`

## 2.10.2 Detect DTMF tones

### **audioDtmfStart()**

**in selected models**

SYNTAX: `audioDtmfStart(portID);`

WHERE: `portID` - port identifier returned by `newAudioPort()`

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Start detection of DTMF tones on the audio input channel.

Tones are passed through the associated local port in data units of type DT\_RS. The "data" element of the data unit holds the recognized key in as the character value. The table below shows how the tones and their respective character value.

<i>pressed key</i>	<i>char value</i>	<i>pressed key</i>	<i>char value</i>
0	0	6	6
1	1	7	7
2	2	8	8
3	3	9	9
4	4	* (star)	10
5	5	# (hash)	11

EXAMPLE: `audioDtmfStart(portID);`

### **audioDtmfStop()**

**in selected models**

SYNTAX: `audioDtmfStop(portID);`

WHERE: `portID` - port identifier returned by `newAudioPort()`

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Stop/hold detection of DTMF tones. Resume detection with `audioDtmfStart()`.

EXAMPLE: `audioDtmfStop(portID);`

## 2.11 Imaging

(in selected models)

Using an optional USB camera, you can take pictures with a single call to `getImage()`. The acquired image is saved to the disk in JPEG format. It can, for example, be shown in a web page or WebGUI, or send it to mobile phones in combination with the MMS feature.

### **getImage()** **in selected models**

SYNTAX: `getImage(camera, filename, width, height [, text [, zoom]] );`

WHERE: camera - number of the camera to grab the image from  
 filename - name (and path) of the target image (extension must be .jpg or .jpeg)  
 width - width of target image in pixels (10 ~ 640)  
 height - width of target image in pixels (10 ~ 480)  
 text - (optional) text message to place over the image  
 zoom - (optional) image zoom factor in percent (100+)

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Acquire an image from an optional USB camera attached to FieldCommander. The acquired image is scaled and, optionally, zoomed in and overlaid with a text message. Up to four camera's can be connected at the same time.

The image is acquired at 640 by 480 pixels. When you scale it down make sure to preserve the aspect ratio of 4:3, or the picture will look deformed.

To save resources and decrease wear on the flash drive, images which are used temporarily are best stored to one of the /tmp directories which are in volatile memory.

EXAMPLE: 

```
// grab photo from camera 1 with 2x zoom
getImage(1, "/www/tmp/photo.jpg", 320, 240, "Camera 1", 200);
```

### **overlayImage()** **in selected models**

SYNTAX: `overlayImage(overlay, target, background [, x-pos, y-pos] );`

WHERE: overlay - name and path of overlay image with transparency (PNG format)  
 target - name and path of the target image (JPEG)  
 background - name and path of the background image (JPEG)  
 x-pos - (optional) horizontal position of background image in pixels  
 y-pos - (optional) vertical position of background image in pixels

RETURN: number - 0 on success, 1 on failure

DESCRIPTION: Merge an overlay image with a background image. This can be used to add a fixed image to a picture such as a logo, or let the picture appear in a decorative frame.

The overlay image should be in true color (32 bit) PNG format. On images with variable levels of transparency, this function will perform full alpha blending. That means that the background can smoothly 'shine through' the overlay. Palette based PNG's (8 bit) are not supported.

The target image will be the size of the overlay image. When the overlay image is smaller than the background, the background will be scaled down. When it is bigger, the background will be placed at the top-left (XY: 0, 0) of the overlay, unless it is explicitly moved using x-pos and y-pos.

To save resources and decrease wear on the flash drive, images which are used temporarily are best stored to one of the /tmp directories which are in volatile memory.

```
EXAMPLE: // add a logo as overlay (320x240) to photo
overlayImage("/data/logo.png", "/www/tmp/result.jpg", "/data/tmp/photo.jpg");

// add a frame as overlay (640x480) and center photo (320x240)
overlayImage("/data/frame.png", "/www/tmp/result.jpg",
"/www/data/photo.jpg", 160, 120);
```

---

## 2.12 Logging

FieldCommander maintains a logfile. In the Log tab in the Configuration manager you can choose the level of information to include/store in this log. Log entries are available at five levels, in order of importance: critical errors, errors, warnings, information and debug messages. See the User's Guide for details.

To ease script debugging, you can add your own log entries by calling the `writeLog()` function. Just provide a message string and a line will be added to the log file with the message, a time stamp. Entries added by `writeLog()` are of type 'information' by default.

### **writeLog()**

---

SYNTAX: `writeLog(message [, level]);`

WHERE: `message` - the text string to write to the FCscript log file  
`level` - (optional), one of the options below:  
LOG\_CRITICAL  
LOG\_ERROR  
LOG\_WARNING  
LOG\_INFO (default)  
LOG\_DEBUG

RETURN: `number` - 0 on success, 1 on failure

DESCRIPTION: Writes a line of text to the log.

```
EXAMPLE: writeLog("this is a critical message", LOG_CRITICAL);
writeLog("the message can contain "+variable+" values");
writeLog("the following:\nis on the next line");
```

---

## 2.13 Calling PHP

Most functions in FCscript explained in this manual are to control the flow of data units or to use one of the services for networking or telecommunication. The FCphp programmers interface features a completely different set of instructions, most of them focussing on reading and writing of the system properties as used for the FieldCommander admin web pages.

In some cases it may be useful to integrate functionality in FCscript which is normally only accessible from a PHP page. To do this, you can run a PHP page from FCscript by calling the `php()` function.

Note that the page should not use any output through `print` or `echo` functions.

### 2.13.1 Passing arguments

You can pass arguments to the PHP page in the "variable=value" form like you normally do when using HTTP GET or POST. The arguments are stored in the `$argv` array in PHP. The example below shows how they can be copied to the `$_GET` array and use them as you would normally do in PHP.

### 2.13.2 Example

FCscript file: `callphp.jse`

```
function main()
{
  php("/www/byscript.php", "name=Tim", "eyecolor=blue", "age=12");
  sleep();
}
```

FCphp file: `byscript.php`

```
<?php
// copy $argv to $_GET array when it is available so the rest of
// the PHP file can access the arguments through $_GET as usual
//
if ($argv)
  for ($i=1;$i<count($argv);$i++)
  {
    $it = split("=", $argv[$i]);
    $_GET[$it[0]] = $it[1];
  }

// write arguments to test.txt for sake of testing
//
$fp = fopen("/home/public/www/test.txt", "a");

while (list($key, $val) = each($_GET)) {
  fprintf($fp, "\t%s=%s", $key, $val);
}

fclose($fp);
?>
```



## 2.14 System functions

### **rebootDevice()**

---

SYNTAX:            `rebootDevice();`

RETURN:            none

DESCRIPTION:      Shutdown and restart FieldCommander. This will cause the FCscript to be aborted. It takes about two minutes before the system is up again. The script calling this function is automatically started when it is back up.

---

### **syncDisk()**

---

SYNTAX:            `syncDisk();`

RETURN:            none

DESCRIPTION:      Force committing of any delayed reads or writes to disk. Normally FieldCommander may delay write actions to save resources and improve overall performance.

---

## Function index

addBuffer()	11
addConverter()	19
addDataTrigger()	18
addExport()	21, 41
addFilter()	16
addGate()	17
addViewpoint()	22
audioDtmfStart()	54
audioDtmfStop()	54
audioSoundPlay()	53
audioSoundStop()	53
clearBuffer()	11
clearMetaValue()	36
exportXML()	34
findFlags()	14
findString()	13
findTime()	15
findValue()	13
getBufferDataElement()	12-15
getBufferDataUnit()	15
getImage()	55
getLocalPort()	9, 10
getMetaValue()	35
getUrl()	27, 32, 33
gotoFirst()	12-15
gotoLast()	12-15
gotoNext()	12-15
gotoPrevious()	13-15
gprsConnect()	38
gprsDisconnect()	38
handleEvent()	11, 18, 27, 28, 40, 51
mailAddAttachment()	31
mailAddRecipient()	30
mailCreate()	30, 31
mailSend()	27, 30-32
main()	10, 32, 57
modemAnswer()	50
modemDial()	50
modemGetCaller()	51
modemHangup()	50, 51
newAudioPort()	7, 10, 53, 54
newLocalPort()	6, 8, 10, 29, 38, 50, 51, 53
newModemPort()	7, 38, 50-52
newRemotePort()	21, 41
newTimer()	24-26, 29
newUsbPort()	29
openPorts()	10, 37, 38
overlayImage()	55
php()	27, 57, 58
pppConnect()	38, 39
pppDisconnect()	39
putUrl()	27, 32, 33
rebootDevice()	59
resetDataTrigger()	18
resetFilter()	16
resetGate()	17
sendMMS()	52
sendSMS()	51
sendString()	29
setFlagCondition()	23
setLocalPort()	6-10, 50, 52, 53
setMetaValue()	36
setStringCondition()	23
setTimeCondition()	23
setValueCondition()	22
sleep()	10, 25, 26, 28, 29, 32
startClockTimer()	26
startIntervalTimer()	25
stopTimer()	24-26
syncDisk()	59
Table commands	
configTableQuery()	48, 49
configTableQueryFinalize()	48, 49
deleteConfigTableRow()	44, 47
getConfigTableRowQuery()	48
getConfigTableRow()	43-46
getFirstConfigTableRow()	44, 45
getLastConfigTableRow()	44, 46
getNextConfigTableRow()	44, 45
getPreviousConfigTableRow()	44-46
setConfigDatabase()	43, 44, 48
setConfigTableRow()	44, 46
tcpConnect()	40-42
tcpDisconnect()	42
tcpExport()	40, 41
tcpSendString()	41, 42
trigger()	27, 28
waitForEvent()	28
writeLog()	56

## Notice to the user

### **IMPORTANT - READ CAREFULLY:**

This CER End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and CER International bv for the CER software product FieldCommander, which includes computer software and may include associated media, printed materials, and "online" or electronic documentation ("SOFTWARE"). The SOFTWARE also includes any updates and supplements to the original SOFTWARE provided to you by CER. Any software provided along with the SOFTWARE that is associated with a separate end-user license agreement is licensed to you under the terms of that license agreement. By installing, copying, downloading, accessing, or otherwise using the SOFTWARE, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, do not install or use the SOFTWARE; you may, however, return it to your place of purchase for a full refund within 10 days of the date you acquired it.

## Software License Agreement

The SOFTWARE is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE is licensed, not sold.

### **GRANT OF LICENSE**

You may install, use, access, display, run, or otherwise interact with ("RUN") one copy of the SOFTWARE, or any prior version for the same operating system, on your FieldCommander. You are obtaining no rights on the SOFTWARE except those given in this limited license.

### **OWNERSHIP**

You may not translate, reverse engineer, decompile, or disassemble the SOFTWARE, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation. You may not rent, lease, or lend the SOFTWARE. You may not modify the SOFTWARE or merge all or any part of the SOFTWARE in another program. The SOFTWARE is licensed as a single product. Its component parts may not be separated for use on more than one COMPUTER.

### **NO TRANSFER**

You may not sublicense the SOFTWARE. You may not transfer the SOFTWARE to a third party unless you cease all use of it, transfer all copies of it and accompanying Documentation, and the transferee agrees to be bound by the terms of this Agreement.

### **TERM**

This License shall continue for as long as you use the SOFTWARE. However, it will terminate if you fail to comply with any of its term or conditions. Upon such termination you must immediately cease using the SOFTWARE and must follow CER International's instructions regarding return of the Software. ALL DISCLAIMERS HEREIN SHALL SURVIVE TERMINATION.

### **LIMITED WARRANTY**

CER International warrants that (a) the HARDWARE will be free from defects in materials and workmanship under normal use and service for a period of one year from the date of receipt; and (b) the SOFTWARE accompanying the HARDWARE will perform substantially in accordance with the accompanying Product Manual(s) for a period of 90 days from the date of receipt. Any implied warranties on the HARDWARE and SOFTWARE are limited to one (1) year and 90 days, respectively. Some states/jurisdictions do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you.

### **CUSTOMER REMEDIES**

CER International's entire liability and your exclusive remedy shall be, at CER International's option, either (a) return of the price paid or (b) repair or replacement of the HARDWARE or SOFTWARE that does not meet the above Limited Warranty. This Limited Warranty is void if failure of the HARDWARE or SOFTWARE has resulted from accident, abuse, or misapplication. Any replacement HARDWARE or SOFTWARE will be warranted for the remainder of the original warranty period or 30 days, whichever is longer.

### **NO OTHER WARRANTIES**

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CER INTERNATIONAL DISCLAIMS ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE ACCOMPANYING PRODUCT MANUAL(S) AND WRITTEN MATERIALS, AND ANY ACCOMPANYING HARDWARE. THE LIMITED WARRANTY CONTAINED HEREIN GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS WHICH VARY FROM STATE/JURISDICTION TO STATE/JURISDICTION

**NO LIABILITY FOR CONSEQUENTIAL DAMAGES**

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CER INTERNATIONAL AND ITS SUPPLIERS SHALL NOT BE LIABLE FOR ANY OTHER DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THIS CER INTERNATIONAL PRODUCT, EVEN IF CER INTERNATIONAL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, CER INTERNATIONAL'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS AGREEMENT SHALL BE LIMITED TO THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE.

**GENERAL**

This License is the entire agreement between us, supersedes any other agreement or discussions, oral or written and may not be changed except by a written signed agreement. CER International accepts no liability for any damages whatsoever arising out of the use of or inability to use this software, direct and/or indirect. This License shall be governed by and construed in accordance with the laws of the Netherlands. If any provision of this License is declared by a Court of competent jurisdiction to be invalid, illegal, or unenforceable, such provision shall be severed from the License and the other provisions shall remain in full force and effect. The parties have requested that this Agreement and all documents contemplated hereby be drawn up in English.

**GOVERNMENT RESTRICTED RIGHTS**

The HARDWARE, SOFTWARE and user documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions. Manufacturer is CER International bv, Roosendaal, the Netherlands. Should you have any questions concerning this Agreement, or if you desire to contact CER International for any reason, please write: Customer Service Dept., CER International bv, PO Box 258, NL 4700 AG Roosendaal, The Netherlands.

## Third-party software

This product contains software under the GNU General Public License and other open source licenses. Copies of these licenses and information about obtaining the GPL source code is available on the web at <http://www.cer.com>. If you would like a copy of the GPL source code contained in this product shipped to you on CD for costs only covering preparing and mailing the CD, contact CER International bv, FieldCommander Product Management, PO Box 258, 4700 AG Roosendaal, The Netherlands.

---

Portions copyright © 1995,1998,1999,2000 by Jef Poskanzer <jef@acme.com>. All rights reserved.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

---

Portions copyright © 1996 - 2001, Daniel Stenberg, <daniel@haxx.se>. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

## Trademarks

CER and FieldCommander are registered trademarks of CER International bv.

All other product names and services identified throughout this book are trademarks or registered trademarks of their respective companies. They are used throughout this manual in editorial fashion only and for the benefit of such companies. No such uses, or the use of any trade name, is intended to convey endorsement or other affiliation with this manual.

## Copyrights

Copyright © 2008 CER International bv. All rights reserved.

No part of this publication may be reproduced in any form, or stored in a database or retrieval system, or transmitted or distributed in any form by any means, electronic, mechanical photocopying, recording, or otherwise, without the prior written permission of CER International bv, except as permitted by the Copyright Act of 1976 and except that program listings may be entered, stored and executed in a computer system.

THE INFORMATION AND MATERIAL CONTAINED IN THIS MANUAL ARE PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY WARRANTY CONCERNING THE ACCURACY, ADEQUACY, OR COMPLETENESS OF SUCH INFORMATION OR MATERIAL OR THE RESULT TO BE OBTAINED FROM USING SUCH INFORMATION OR MATERIAL, NEITHER CER INTERNATIONAL BV NOR THE AUTHORS SHALL BE RESPONSIBLE FOR ANY CLAIMS ATTRIBUTABLE TO ERRORS, OMISSIONS, OR OTHER INACCURACIES IN THE INFORMATION OR MATERIAL CONTAINED IN THIS MANUAL, AND IN NO EVENT SHALL CER INTERNATIONAL BV OR THE AUTHORS BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF SUCH INFORMATION OR MATERIAL.

from Device to Enterprise

**CER**<sup>®</sup>

The latest documentation is available  
from <http://www.cer.com>

FieldCommander is a product of:

CER International bv  
Postbus 258  
NL 4700 AG Roosendaal  
The Netherlands  
TEL: +31 (0)165 557417  
FAX: +31 (0)165 562151  
<http://www.cer.com>

Part no. FCSCRPG, revision 3.5.31